Automatic Test Suite Generation

for Scientific MATLAB Code

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Venkata Subhash Movva

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Dr. Andrew Brooks

July 1, 2015

Acknowledgements

I would like to thank my advisor Dr. Andrew Brooks for his constant support and guidance throughout the course of the thesis.

I would like to thank my committee members Dr. Gary Shute and Dr. Steven Trogdon for evaluating my thesis and for their valuable suggestions.

I also would like to thank Dr. Pete Willemsen, Dr. Hudson Turner, Dr. Ted Pedersen, and Dr. Wang for sharing their valuable knowledge. I also thank Lori Lucia and Clare Ford for all their support and help.

I am grateful for all the encouragement and guidance from my cousin Y. Chaitanya. I also would like to thank my sister M. Gowthami and my brother-in-law P. Aditya for their support and encouragement.

Dedication

This thesis is dedicated to my parents, Koteswara Rao Movva and Madhavi Latha Movva and to my grandmother Parvathi Yalavarthy. I would also like to dedicate my thesis to my sister Gowthami Movva, my brother-in-law Aditya Polumetla, and to my cousin Chaitanya Yalavarthy.

Abstract

Software testing is the process of finding code faults by applying tests and comparing results from the code to an oracle. Mutation testing is one of many testing techniques. A mutation is a single syntactic change to the original code. A mutation score is the percentage of mutants detected by any given test suite. So it is possible to compare the effectiveness of different test suites.

Testing techniques cannot be easily applied to scientific code for two reasons. First, an oracle is usually unavailable. Second, scientific code output typically deals with real numbers rather than whole numbers. Correctness of the code depends on the tolerance level that is acceptable. Mutation sensitivity testing tackles the tolerance problem by systematically exploring what happens across a range of relative error between a mutation and the original program under test.

This thesis is an extension to earlier work on mutation sensitivity testing of scientific MATLAB code. An automatic test case generation technique is proposed based on the use of a genetic algorithm. This approach allows for the creation of test suites which detect mutants at the highest possible levels of relative error. Test suites have been automatically generated for the 8 scientific functions used in earlier work and comparisons drawn with the results from existing manual test suites. As a final step, the 8 scientific functions were unit tested by using independent technologies to calculate expected outputs from the generated test inputs.

Table of Contents

List of Tables ..... **vi**

List of Graphs ..... **vii**

List of Figures ..... **viii**

1 Introduction **1**

2 Mutation Testing **3**

2.1 Introduction to Mutation Testing 3

2.2 Mutation Testing Process 4

2.3 Mutation Operators 8

2.4 Selective Mutation 13

2.5 Higher Order Mutation Testing 16

2.5.1 Second Order Mutants 18

2.5.2 Subsuming HOM's 18

3 Mutation Sensitivity Testing & Replication Study **19**

3.1 Challenges in Testing Scientific Software 19

3.2 Limitations of Traditional Mutation Testing 20

3.3 Introduction to Mutation Sensitivity Testing 22

3.3.1 Mutant Detection in Mutation Testing 22

3.3.2 Mutant Detection in MST 22

3.3.3 Maximum Exhibited Error 23

3.4 MATmute (Mutation Sensitivity Testing Tool) 23

3.4.1 MATmute Installation 24

3.4.2 Python Module 26

3.4.3 MATLAB Module 29

4 Implementation **33**

4.1 Drawbacks of Popperian and Pseudo-Random Testing 33

4.1.1 Popperian Testing 33

4.1.2 Pseudo-Random Testing 34

4.2 Automated Effective Test Suite Generation 34

4.2.1 Manual Generation of Test Cases 38

4.2.2 Genetic Algorithm 43

4.2.3 Automated Test Suite Generation Using Genetic Algorithm 44
4.2.4 Independent Oracle Creation and Unit Testing 56
5 Results 69
5.1 Comparison between Hook's test cases and the auto generated test cases 85
5.2 Generating test cases for a function in LUCY package 86
6 Conclusions and Future Work 87
6.1 Conclusions 87
6.2 Future work 87
Bibliography 89

List of Tables

Table 1: Mothra Mutation Operators 13
Table 2: Non-selective mutation scores for 2-selective mutation test sets 15
Table 3: Savings obtained by 2-selective Mutation 16
Table 4: Example of Higher Order Mutant 17
Table 5: Comparison between Hook's and Tauto test suites 85

List of Graphs

Graph 1: Example output graph from MATmute 25
Graph 2: Example of detection graph 32
Graph 3: Mutant detection graph for sphereFnet function 35
Graph 4: A 3D graph to find a test case for the nwtsqrt mutant 42
Graph 5: sphereFnet detection graph generated using Hook's test suite 70
Graph 6: sphereFnet detection graph generated using auto generated test suite 71
Graph 7: simpson detection graph generated using Hook's test suite 72
Graph 8: simpson detection graph generated using the auto generated test suite 72
Graph 9: powerit detection graph generated using Hook's test suite 74
Graph 10: powerit detection graph generated using auto generated test suite 74
Graph 11: odeRK4 detection graph generated using Hook's test suite 76
Graph 12: odeRK4 detection graph generated using auto generated test suite 76
Graph 13: nwtsqrt detection graph generated using Hook's test suite 78
Graph 14: nwtsqrt detection graph generated using auto generated test suite 78
Graph 15: GEPiv detection graph generated using Hook's test suite 80
Graph 16: GEPiv detection graph generated using auto generated test suite 80
Graph 17: gaussQuad detection graph generated using Hook's test suite 82
Graph 18: gaussQuad detection graph generated using auto generated test suite 82
Graph 19: binSearch detection graph generated using Hook's test suite 84
Graph 20: binSearch detection graph generated using the auto generated test suite 84
Graph 21: adjextent detection graph generated using auto generated test suite 86

List of Figures

Figure 1: Mutation Testing Process 5
Figure 2: Design and working of Pymute.....29
Figure 3: Design and working of MATmute.....31
Figure 4: Test suite generation process 45

1 Introduction

Computational science is a field in which computer science concepts are applied to advance real-world science. Computational science has a very broad set of domains, such as aerospace engineering, nuclear engineering, physics, biology, chemistry, and mechanical engineering. Because of the complexity of computational software, scientists must test their code well before publishing it. There were several cases in the scientific community where published papers have been retracted due to the detection of previously undiscovered code faults. Software engineering concepts cannot be applied to scientific code for two reasons. The first reason is the lack of oracles for computational software. The second reason is the tolerance problem.

Scientists rather than only test their science must also test their code to detect code faults. To test code efficiently, a scientist should have an efficient test suite. Mutation testing tests the test suite by generating a mutation score. If the mutation score is high enough, then the test suite can be considered as effective. However, the mutation testing approach doesn't address the tolerance problem. Mutation sensitivity testing applies mutation testing to scientific code by using tolerances instead of strict equality.

This thesis is an extension to mutation sensitivity testing (MST)[16]. In MST the author proposes the creation of manual test cases by analyzing the code under test. This is time consuming. The author of MST also assumes that the tester has access to oracles. While testing the code, it is usually difficult for scientists to know the exact tolerance value they

should be working to. In this thesis, I tried to solve all the above mentioned problems. I propose to generate test cases that detect mutants at high relative error. Test cases that detect mutants at high relative error are generated automatically using a genetic algorithm. Oracles for MATLAB code are created using independent technologies such as Java and R.

Chapter 2 gives a detailed view on mutation testing. Chapter 3 describes mutation sensitivity testing and the working of the MATmute tool. Chapter 4 describes automatic test case generation and the unit testing process. Chapter 5 presents the results calculated for 8 MATLAB functions. Chapter 6 presents the conclusion and ideas for future work.

2 Mutation Testing

This chapter explains the mutation testing process, mutation operators, selective mutation and higher-order mutation.

2.1 Introduction to Mutation Testing

Mutation testing is the process of measuring the effectiveness of a test suite in detecting code faults. Test suite quality is measured in terms of a mutation score. If the mutation score is high then test suite effectiveness is considered high.

The main idea of mutation testing is to replicate the real world programming mistakes made by programmers in the form of mutants. A mutant is a replica of the code under test but with a single syntactic change. Many mutants with a different syntactic change are created for the code under test. The effectiveness of the test suite is measured by executing these mutants against the test cases. The mutant is said to be detected if the result of the mutant is different from the result of original program for any test case in the test suite. After processing all the mutants, the number of mutants detected are found. The number of detected mutants divided by the total number of mutants gives the mutation score.

The mutation testing concept was first proposed by a student, Richard Lipton, in the year 1971. Later, the idea was extended by DeMillo and Hamlet in the late 1970s. Timothy

Budd in 1980 developed the first mutation tool as part of his PhD research work. Mutation testing requires more computing power to execute the many mutants. Due to lack of computing power in the late 20th century, mutation testing was not popular and was not used widely. Nowadays, with the tremendous increase in computing power, more research work is being done on mutation testing to increase its efficiency and effectiveness.

2.2 Mutation Testing Process

The mutation testing process is shown in Figure 1. The process of mutation testing is described below step by step.

1. Generate mutants P' by making a single syntactic change to the original program P. Various mutation operators can be used to generate the mutants. Different types of mutation operators are explained later in this chapter.
2. Create test suite T to test the original program. Perform unit testing on the original program using all the test cases to check the correctness of the program.
3. If the original program P is not correct and fails a unit test, then P should be corrected before going further in the process.
4. If the original program P looks correct, then run all the test cases in T on all mutants P'. If the P' result is different to the P result on at least one test case, then P' is said to be detected. The mutation score is produced in this stage by dividing the total number of detected P' with the total number of P'.

5. If most or all of the mutants (P') are detected and the mutation score is satisfactory, then it means that the test suite T contains quality test cases. Now the process is complete.
6. If a significant number of mutants are not detected and the mutation score is not satisfactory, then undetected mutants are analyzed and new test cases are added to the test suite. This process is repeated until a satisfactory mutation score is achieved for the test suite. Equivalent mutants which are syntactically different when compared to the original program P but functionally the same are also a reason for undetected mutants. They always produce the same output as P and are impossible to detect automatically. These mutants should be detected and removed manually by the tester.

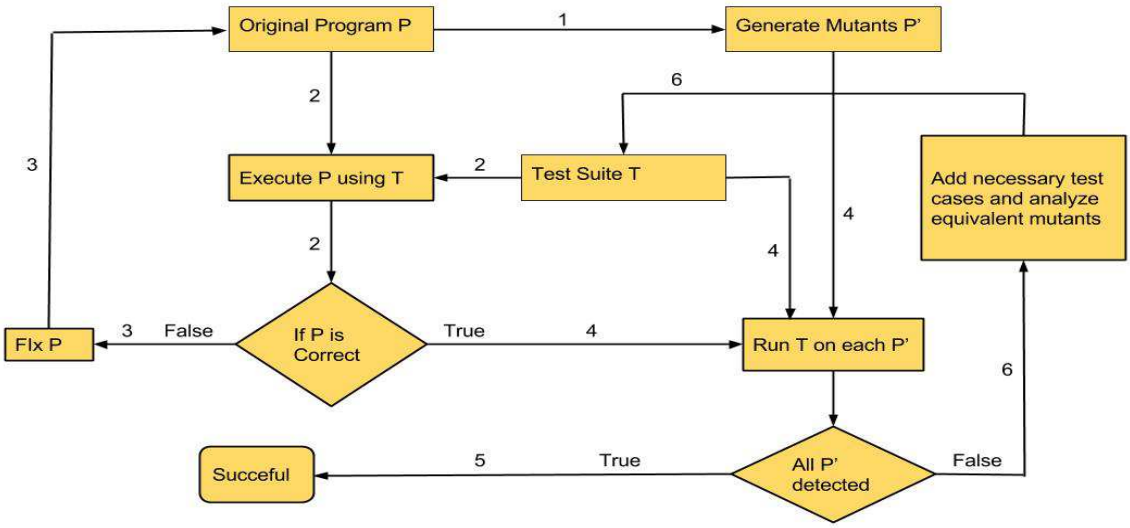


Figure 1: Mutation Testing Process (Numbers on the arrows represent the step numbers in the process)

Example of mutant:

Original Program

```
function result = greater(a, b)
    if (a > b)
        result = a;
    else
        result = b;
end
```

Mutant

```
function result = greater(a, b)
    if (a <= b)       // Mutation change >  to <=
        result = a;
    else
        result = b;
end
```

Example of equivalent mutant:

Original Program

```
function result = functionX (a, b)
    ……
    ……
    If (a = = 2 && b = = 2)
        result = a+b;
    ……
End
```

Equivalent Mutant

```
function result = functionX (a, b)
    ……
    ……
    If (a = = 2 && b = = 2)
        result = a*b;
    ……
end
```

2.3 Mutation Operators

Transformation rules are required to generate mutants by making a single syntactic change to the original program. These transformation rules which generate mutants from the original program are called mutation operators. In the papers [1], [2], and [3] many mutation operators are discussed. For this thesis, the mutation operators described in [4] are used. The following are the mutation operators used by the MATmute tool [16] and their description.

- Statement Deletion: This mutation operator deletes a line of code.

Example for Statement Deletion:

Original Program

```
function result = greater(a, b)
    if (a > b)
        result = a;
    else
        result = b;
end
```

Mutant

```
function result = greater(a, b)
    if (a > b)
        result = a;
    //else  // This statement is deleted
        result = b; end
```

- Conditional Negation: The conditional part in an If or While statement is negated by this mutation operator.

Example for Conditional Negation:

Original Program

```
function result = greater(a, b)
    if (a > b)
        result = a;
    else
        result = b;
end
```

Mutant

```
function result = greater(a, b)
    if ~(a > b) // statement negated
        result = a;
    else
        result = b;
end
```

- Constant Replacement: This mutation operator changes a hard-coded constant C to 0, -C, C-1, C+1, 0.9C, 1.1C, and results in six mutants per hard-coded constant.

Example for Constant Replacement:

Original Program

```
function result = circleArea(radius)

    Pi = 3.14;
    result = Pi * radius * radius;

end
```

Mutant

```
function result = circleArea(radius)

    Pi = 4.14; // constant changed 3.14 ->4.14
    result = Pi * radius * radius;

end
```

- Operator Replacement: This mutation operator deals with arithmetic operators (+, -, *, /, \, ^), relational operators (<, <=, >, >=, ==, ~=), and logical operators (&& and ||). Replaces any of the three operators (arithmetic, relational, and logical) with another operator of the same class. An arithmetic operator results in five mutants, a relational operator results in five mutants, and a logical operator results in one mutant.

Example for Operator Replacement:

Original Program

```
function result = circleArea(radius)


    Pi = 3.14;
    result = Pi * radius * radius;


end
```

Mutant

```
function result = circleArea(radius)


    Pi = 3.14;
    result = Pi + radius * radius; // operator changed * -> +


end
```

Mutation Operator	Description
AAR	array reference for array reference replacement
ABS	absolute value insertion
ACR	array reference for constant replacement
AOR	arithmetic operator replacement
ASR	array reference for scalar variable replacement
CAR	constant for array reference replacement
CNR	comparable array name replacement
CRP	constant replacement
CSR	constant for scalar variable replacement
DER	DO statement end replacement
DSA	DATA statement alterations
GLR	GOTO label replacement
LCR	logical connector replacement
ROR	relational operator replacement
RSR	RETURN statement replacement
SAN	statement analysis
SAR	scalar variable for array reference replacement
SCR	scalar for constant replacement
SDL	statement deletion
SRC	source constant replacement
SVR	scalar variable replacement

UOI	unary operator insertion

Table 1: Mothra Mutation Operators (operators used in MATmute are in bold)

Table 1 lists the Mothra mutation operators, the most widely used mutation operators in mutation testing. The 22 mutation operators in Mothra set [10] were derived by carefully researching the simple errors that programmers make. This set of mutation operators was obtained after 10 years of refinement through various mutation testing systems.

2.4 Selective Mutation

Mutation testing is a costly computational testing technique. The major cost is incurred when mutants are executed against the test suite. To reduce computational cost and to make mutation testing efficient, many cost reduction techniques have been proposed. Out of many cost reduction techniques, selective mutation is widely used.

Selective mutation reduces the computational cost by reducing the number of mutant programs. As the mutant programs are generated from mutation operators, eliminating certain operators will reduce the number of mutant programs. The main idea of selective mutation is to find a small set of mutation operators that generate a subset of all possible mutants with no loss of test effectiveness. The idea of selective mutation was first proposed by Mathur[8].

Some operators generate more mutants than others. More mutants results in redundancy

and also increases the computational cost of testing. The idea of "2-selective mutation" was proposed and implemented by Offutt [9] to overcome this problem. 2-selective mutation is the process of eliminating the two mutation operators that generate the most number of mutants.

Let us look at the steps of an experiment performed by Offutt [9] to get a clear idea on 2-selective mutation:

1. Ten Fortran-77 programs ranging from 10 to 48 executable statements were chosen for the experiment.
2. From the 22 Mothra mutation operators, ASR and SVR were identified as operators that generate the most number of mutants. These two mutation operators were left out.
3. Mutants were created using the remaining 20 mutation operators, and equivalent mutants were removed.
4. Test cases were developed to kill the non-equivalent mutants. Mutation scores are recorded for each test suite.
5. The same process was repeated for non-selective mutation (no operators eliminated). Test cases are developed and their mutation scores are recorded.
6. To measure the effectiveness of selective mutation, test sets developed for selective mutation were run against the non-selective mutants and mutation scores were computed. The mutation scores showed that the test cases that were 100% adequate for 2-selective mutants were almost 100% adequate for non-selective

mutants. Also, the number of mutants for selective mutation is observed to be very much less when compared to non-selective mutation. Tables 2 and 3 show the results for Offutt's experiment.

Program	Test Cases	Number Live Mutants	Mutation Score
Banker	57.4	0.0	100.00
Bub	7.2	0.0	100.00
Cal	50.0	0.0	100.00
Euclid	3.8	0.0	100.00
Find	14.6	0.6	99.94
Insert	3.8	0.0	100.00
Mid	25.2	0.0	100.00
Quad	12.2	0.0	100.00
Trityp	44.6	0.2	99.98
Warshall	7.2	0.0	100.00
Average	22.6	0.1	99.99

Table 2: Non-selective mutation scores for 2-selective mutation test sets

Program	Non-Selective Mutants	Selective Mutants	Percentage Saved
Banker	2765	1944	29.69
Bub	338	277	18.05
Cal	3010	2472	17.87
Euclid	196	142	27.56
Find	1022	622	39.15
Insert	460	352	23.48
Mid	183	133	27.32
Quad	359	279	22.28
Trityp	951	810	14.83
Warshall	305	259	15.08
Total	9589	7290	23.98

Table 3: Savings obtained by 2-selective Mutation

2.5 Higher Order Mutation Testing

Higher order mutation testing is a way for generating mutants by applying more than one mutation operator at a time. This concept was first proposed by Offut [11] in 1992. By using this technique, the number of mutants can be reduced by 50% or more without loss of effectiveness of mutation testing. It is also observed from papers [12] and [13] that the

number of equivalent mutants can be reduced from about 18.66% to 5.00% and that generated higher order mutants can also be harder to kill. Table 4 gives an example of higher order mutant.

Program	First Order Mutant 1 (FOM1)

…
while((str[i] == str[j]) && (j >= 0)) {
i++;
j--;
}
...
	…
	while((str[i] <= str[j]) && (j >= 0)) {
	i++;
	j--;
	}
	...

First Order Mutant 2 (FOM2)	**Higher Order Mutant (FOM1 + FOM2)**

…
while((str[i] == str[j]) && (j <= 0)) {
i++;
j--;
}
...
	…
	while((str[i] == str[j]) && (j >= 0)) {
	i++;
	j--;
	}
	...

Table 4: Example of Higher Order Mutant

2.5.1 Second Order Mutants

In [13], Polo showed that higher order mutants do not decrease the quality of the a suite. Polo conducted his experiments on second order mutants (mutants with 2 faults) and proposed three algorithms to generate them. The first algorithm is the LastToFirst algorithm. Assume we have n first order mutants (FOMs). SOMs are generated by bringing together the first FOM and FOM number n, the second FOM and FOM number n-1, and so on. The second algorithm is the DifferentOperators algorithm. SOMs are created by combining FOMs generated by different mutation operators. Lastly, in the RandomMix algorithm, any 2 random FOMs are combined to generate SOMs. Details of other interesting algorithms to generate SOMs are available in [15].

2.5.2 Subsuming HOM's

The concept of "subsuming HOMs" was proposed by Jia and Harman in [14]. An HOM is called a subsuming HOM when it is harder to kill than its FOMs. Harder to kill HOMs will result in generating good test suites. They also suggested some approaches to find subsuming HOMs using a greedy algorithm, a genetic algorithm, and a hill-Climbing algorithm. Jia and Harman, after conducting several experiments with 10 C programs, came to the conclusion that a genetic algorithm approach is the most efficient for finding subsuming HOMs.

3 Mutation Sensitivity Testing & Replication Study

This chapter discusses the limitations of mutation testing and describes how these limitations can be rectified by mutation sensitivity testing [16]. The MATmute tools internal working is explained and information on how to use MATmute effectively is given.

3.1 Challenges in Testing Scientific Software

Research in software engineering has been mainly focused on software engineers and there has been very little research in the field of computational science and engineering. Testing scientific software has some unique challenges which are typically not addressed by the software engineering literature.

The first challenge is the lack of testing oracles. An oracle is an external source which generates expected results for the program under test. In the software engineering literature, all the testing techniques expect the developers to have good oracles. On the other hand, computational science software developers will most probably be developing something very new and have no access to oracles.

The second challenge is the tolerance problem. Most scientific program output exhibits both acknowledged and expected errors that are unavoidable. Some examples of

acknowledged errors are roundoff errors and errors that are introduced when finite precision numbers are used for calculations. This tolerance problem is typically neglected by the software engineering literature.

Due to oracle and tolerance limitations, it is very difficult to determine the exact output for a given test case. So while testing scientific software, testers must depend on estimates of expected output and estimates of tolerance values. It is extremely difficult to calculate the acknowledged errors and to predict the tolerance value. For a mutant to be detected, it should exhibit an error greater than the tolerance value. In this thesis, we will look at a way to choose tests that generate relative errors far from any estimated tolerance value.

3.2 Limitations of Traditional Mutation Testing

Traditional mutation testing deals with the oracle problem by considering the program under test as the oracle [16]. The research done on mutation testing until now assumes that the mutant is killed when the output of the mutant is strictly not equal to the output of the original program. The following example highlights why the strict equality rule is not good to depict the quality of test cases if floating-point numbers are used in the code.

Assume this line of code is a line in the program under test:

```
y = 1 - tan(x * 2 * pi)
```

Assume the mutated line of code is:

```
y = 1 - tan(x * 1 * pi)
```

If the test case used is x = 1, then the following are the results from the 64-bit MATLAB platform for the above two statements.

Original code: y = 1 + 2.4493e-16

Mutant result: y = 1 + 1.2246e-16

Note that the correct result for both the above lines of code is y = 1. But neither of the calculations are strictly correct. So, in this case, the mutant is considered to be killed. In the above case, both lines of code should be equal, but because of strict equality the mutant is killed and this may lead to having a false confidence on the test suite. This situation can be fixed if the mutant exhibits a big error by using a different test case such as x = 0.3.

As is evident from the above example, strict equality is not suitable for mutation testing. Hook in the paper “Mutation Sensitivity Testing” [16] gave a solution for this problem. Hook proposed to use tolerances instead of strict equality.

3.3 Introduction to Mutation Sensitivity Testing

The mutation sensitivity testing (MST) concept was proposed by Hook [16] to address the problems of mutation testing when applied to scientific code. MST solves the problem of strict equality by calculating the relative error between the mutant and the original program output.

3.3.1 Mutant Detection in Mutation Testing

$P_m(t) = P_o(t)$

P_m = Mutant program

P_o = Original program

t = test case

3.3.2 Mutant Detection in MST

$$\gamma(P_m, t) = \frac{|P_m(t) - P_o(t)|}{|P_o(t)|}$$

γ = Relative error

P_m = Mutant program

P_o = Original program

t = test case

If P_m terminally fails then the relative error is equal to infinity. If the relative error result overflows for a legal value of $P_m(t)$, then the relative error will be set to the largest double precision number ($1.7977 * 10^{308}$) (in MATLAB 64-bit).

3.3.3 Maximum Exhibited Error

$$\Gamma(P_m, T) = max_{t \in T}\{\gamma(P_m, t)\}$$

Γ = largest relative error

T = Test suite

γ = Relative error

P_m = Mutant program

P_o = Original program

t = test case

Γ is the largest relative error exhibited by a mutant Pm for a given test suite T.

3.4 MATmute (Mutation Sensitivity Testing Tool)

Many code mutators have been developed for software engineers, As far as we know, there is just one tool (MATmute) which was developed for computational scientists. MATmute was developed by Hook and was demonstrated at CASCON 2008. MATmute generates mutants for MATLAB code by using the mutation operators described in Chapter 2 and it also gives the mutation score for the test suite used.

MATmute has two main components, a Python module to generate mutants (.m files) and the second is a MATLAB module to run these mutants and generate mutation sensitivity scores. Finally, a comparison graph is generated by the tool to display the mutation scores for respective test cases. Detailed explanations of these modules are given below.

3.4.1 MATmute Installation

MATmute is free software. It requires Python 2.5 or 2.6 and MATLAB to run on a system. It can be downloaded from the following link:
http://matmute.sourceforge.net/download.html.

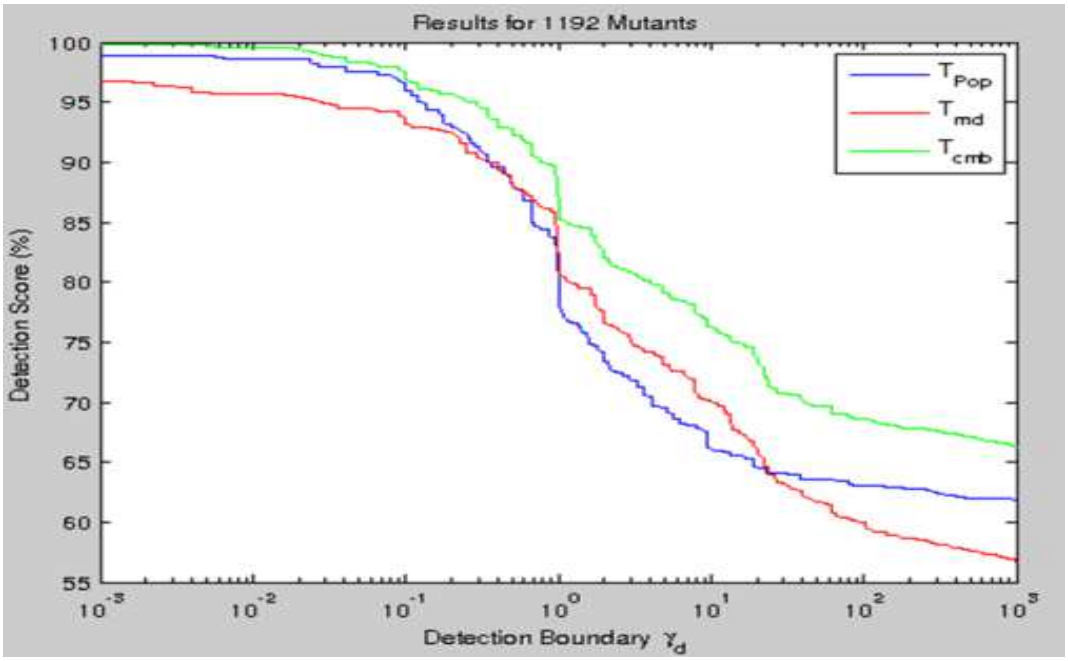
After downloading MATmute and unzipping it, two folders matlab and pymute can be seen in the src folder. In order for MATmute to work, the matlab directory should be added to MATLAB path, and the pymute directory should be added to the Python path. Adding paths should be done within MATLAB as it has its own PYTHONPATH variable. The first line in the script below adds the matlab directory to the search path, and the second line sets the pymute directory to the PYTHONPATH variable.

path(path,' matlab directory path')
setenv('PYTHONPATH', ' pymute directory path')

After executing the above script, we should be able to use MATmute. The above script should be executed whenever MATLAB is restarted. To test whether MATmute is installed properly and is working fine, run test_script.m which can be found in the examples folder. Follow the screen capture below to run test_script.m with default options.

```
>> test_script
Workspace will be cleared, is that ok? (y/n): y

Do you want routines to execute in verbose mode? (y/n): n

Currently the default mutation operators are 'sdl', 'crp', 'neg', and 'orp'.
The 'asp' operator can also be used.
Specify the mutation operators that should be used or press return for the default set:


By default all 8 functions are tested.
Specify the functions that should be tested or press return for the default set:


Loading sets of tests into TPop and Trnd structs.
```

If MATmute is working fine, there must be no errors while executing test_script and the graph below should pop up.



Graph 1: Example output graph from MATmute

3.4.2 Python Module

The Python module named pymute takes a matlab function (.m file) as input and generates mutants for the function. The generated mutants are stored in a folder which is created when pymute is executed. Pymute consists of 6 files or sub modules and 738 lines of code. The 6 sub-modules in pymute are:

1. MATmute : Main file
2. Mutatee : Code file that has to be mutated gets cleaned.
3. Operators : It stores the mutation operators that has to be applied on code.
4. Mutants : Creates mutant files with the help of Mutator.
5. Ops_config: Mutation operators are defined in this file.
6. Mutator : Takes code and mutation operators and will generate mutations.

Pymute can be called from the command line using:

→ **python - m matmute 'function name'**

3.4.2.1 Working of Pymute

The working of pymute is described in steps below. Figure 2 gives an easy to follow overview of these steps.

1. MATmute is the main file and it controls the flow. Initially Mutatee is called to clean and store each statement of the target code. The code here is instrumented (see below) and then stored in a list.

2. MATmute calls Operators to get the set of mutation operators that have to be applied on the target code.
3. MATmute gives Operators and the Mutatee object to Mutants, and asks it to get all possible mutations of the code.
4. Mutants takes Mutator help to generate mutants. Mutator calls operators to generate mutants.
5. Operators generates the mutations as configured in Ops_config.
6. Finally, MATmute asks Mutants to generate mutant files and store them in a separate directory.

3.4.2.2 Instrumented Code

To produce mutants the target is instrumented first and then this instrumented code is used to generate mutants. Instrumentation removes comments, breaks multi-statement lines, and inserts code to monitor loops. The steps for the code instrumentation process are given below:

1. Initially, scan the target code for block comments (statements within '%{', and '%}') and remove these comments.
2. Remove remaining comments such as line comments ('%').
3. Remove line continuations ('...'), and reformat the text so that there is just one statement per line.
4. Store the instrumented code into a list in such a way that each statement is stored

as an item in the list.

5. Search the list for “while” and “for” loops, and add loop monitoring (see below) code for them.
6. Find statements that should not be mutated in the target code, such as function declarations. Add mutation flags around these statements.
7. Remove blank lines and trailing spaces.

3.4.2.3 Loop Monitoring

Mutants generated can introduce faults that lead to an infinite loop. These faults are not traceable and very hard to find. To address this problem, pymute adds loop monitoring code to the instrumented code. MATmute records the number of times the loops in the instrumented code are executed and assigns this number to tick_limit (global variable). Then tick_limit is multiplied with a value greater than 1.

Loop monitoring code increments the loop count by 1 at the start of every loop iteration and then checks whether loop count exceeds the tick_limit or not. If the loop count exceeds the tick_limit, then the mutant terminates itself and records a relative error of infinity for that test-mutant pair.

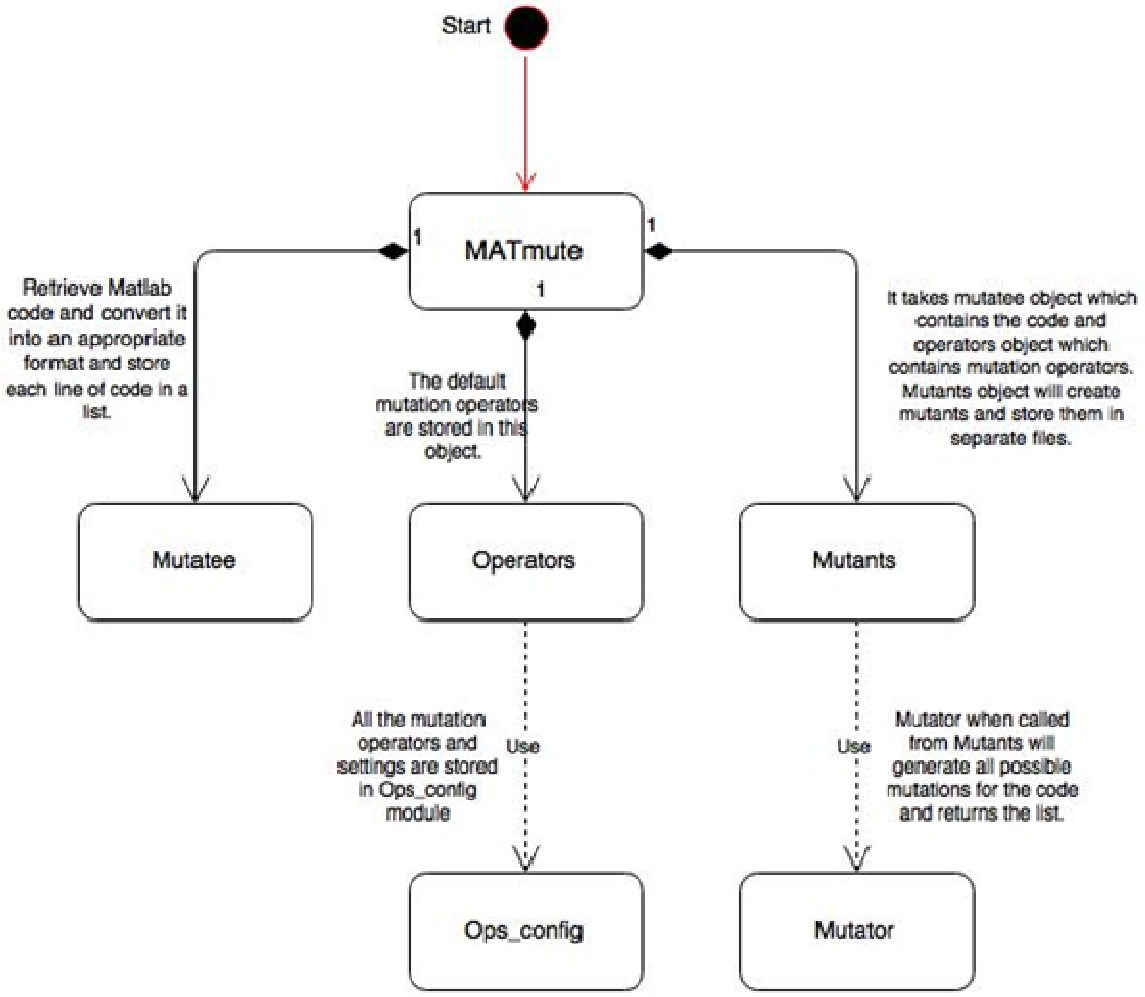


Figure 2: Design and working of Pymute

3.4.3 MATLAB Module

The MATLAB module takes the test suite and MATLAB function as input and gives the mutation score as output. Though pymute can be called from the command line, the MATLAB module must be called from a MATLAB environment. This module executes the mutants generated by pymute against all the test cases in the test suite, and then using these values produces a mutation sensitivity graph or mutation detection graph. Three

modules or files are used in this module; they are Test_script, Matmute, and Get_error. Figure 3 provides an overview of the working of this module. A step by step explanation of the working of the MATLAB module is given below:

1. Initially, a test suite and function are given as input to the Test_script.
2. Test_script calls Matmute by sending it the function and test suite.
3. Matmute calls pymute to generate mutants.
4. Matmute runs all the test cases on the original function and stores the results. The test cases are also run on the instrumented code just to check whether the output of the original function and instrumented code matches. If it does not match, then it gives an error saying that instrumentation was not done properly.
5. Matmute executes all the mutant-test pairs and calculates the relative error between the mutant's output and the original result.
6. Matmute calls Get_error to get the relative error.
7. All the relative errors for each mutant-test pair are calculated and stored in an errors list. Matmute sends this relative errors list to Test_script. The relative errors list is a 2-D array where rows represent mutants and columns represent test cases.
8. In Test_script, equivalent mutants and terminal failure mutants are detected and their values are removed from errors list. If a mutant has a relative error of 0 for each test case then that mutant is identified as an equivalent mutant. And if a mutant has a relative error of infinity for each test case, then that mutant is considered a terminal failure mutant.

9. Maximum relative errors are then identified for each mutant. Test_script calculates the mutation score and plots the detection graph. In the detection graph, the X-axis represent the mutant detection boundary and the Y-axis represents the mutation score. Graphs for different test suites can be generated at a time and details of the test suites will be displayed in the legend of the graph. An Example of a detection graph can be seen in Graph 2 below.

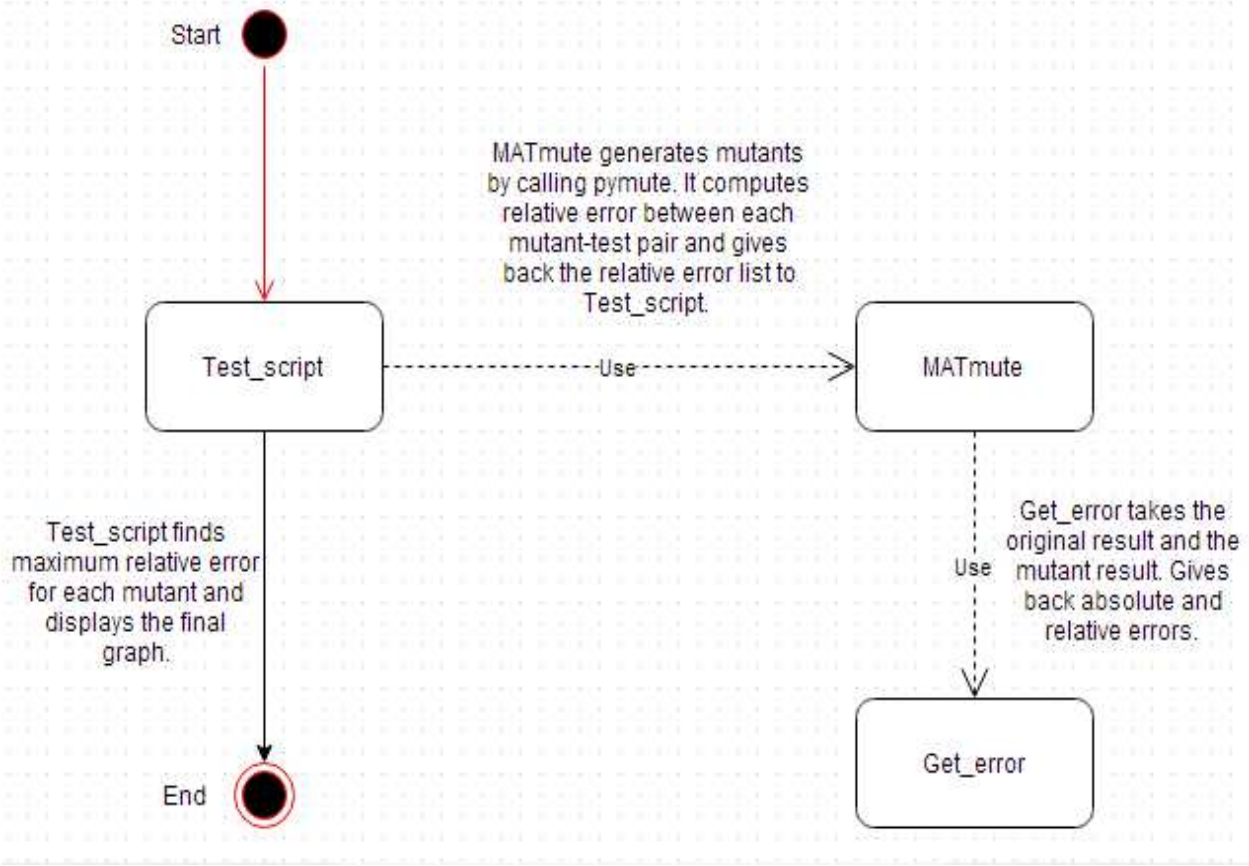
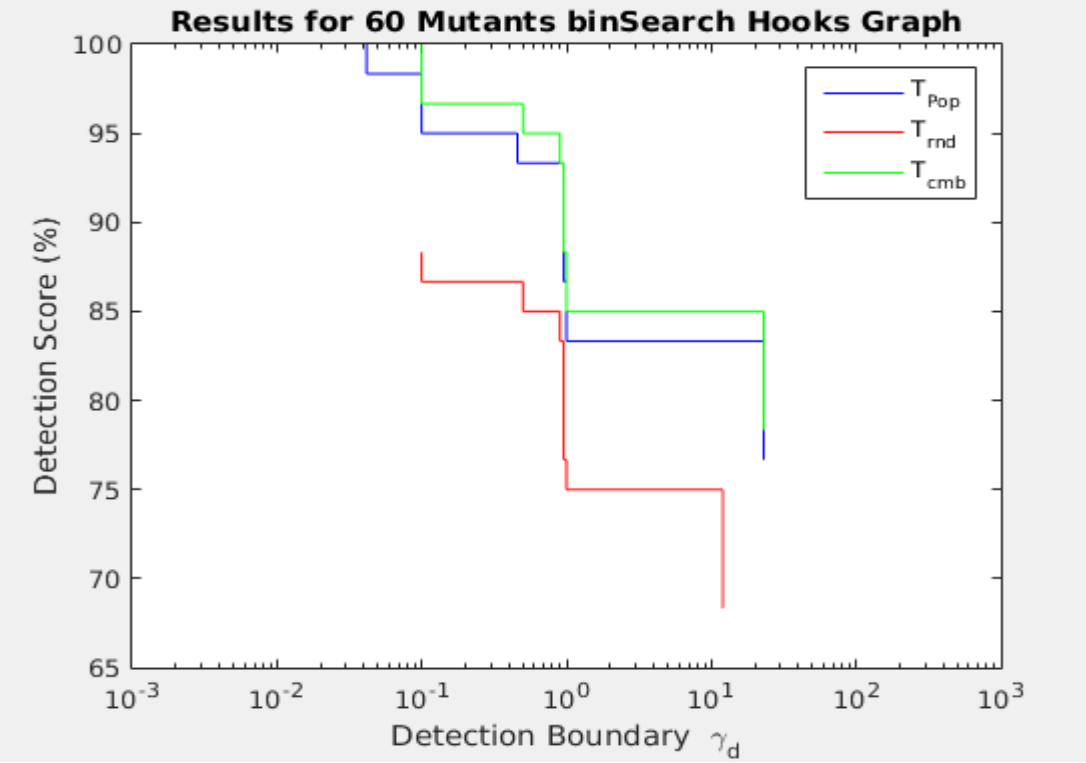


Figure 3: Design and working of MATmute



Graph 2: Example of detection graph

4 Implementation

In this thesis we propose a new way to find test cases for a function or code under test. Hook in his thesis [16] used popperian testing and pseudo-random testing to create test cases for a function. This chapter discusses the limitations of popperian and pseudo-random testing and explains how to overcome these limitations using the automatic test suite generation proposed. Also the oracle problem of mutation testing has been resolved in this thesis by using independent technologies to create oracles.

4.1 Drawbacks of Popperian and Pseudo-Random Testing

This thesis extends the work on mutation sensitivity testing by Hook. Popperian and pseudo-random testing were used by Hook. However, these testing techniques were not automatic and were time consuming. Let us look at each of these testing techniques.

4.1.1 Popperian Testing

This technique is a mix of both boundary value testing and equivalence class partitioning testing. In boundary value testing the test case boundaries for the function under test are predicted. Then test cases are chosen such that they are just near to the boundaries and also in between the boundaries. In equivalence class partitioning testing input domain regions must be determined and test cases must be selected for each region. Popperian testing is proposed as a way to find test cases for computational science software. This testing technique is loosely based on Karl Popper's falsification idea. Popperian testing is

a process of finding test cases that push the boundaries of a program to falsify the code. However, boundary testing inputs are not useful when they give nonsense outputs such as NaN (Not a Number) or Inf (Infinity). These type of inputs are removed from popperian tests.

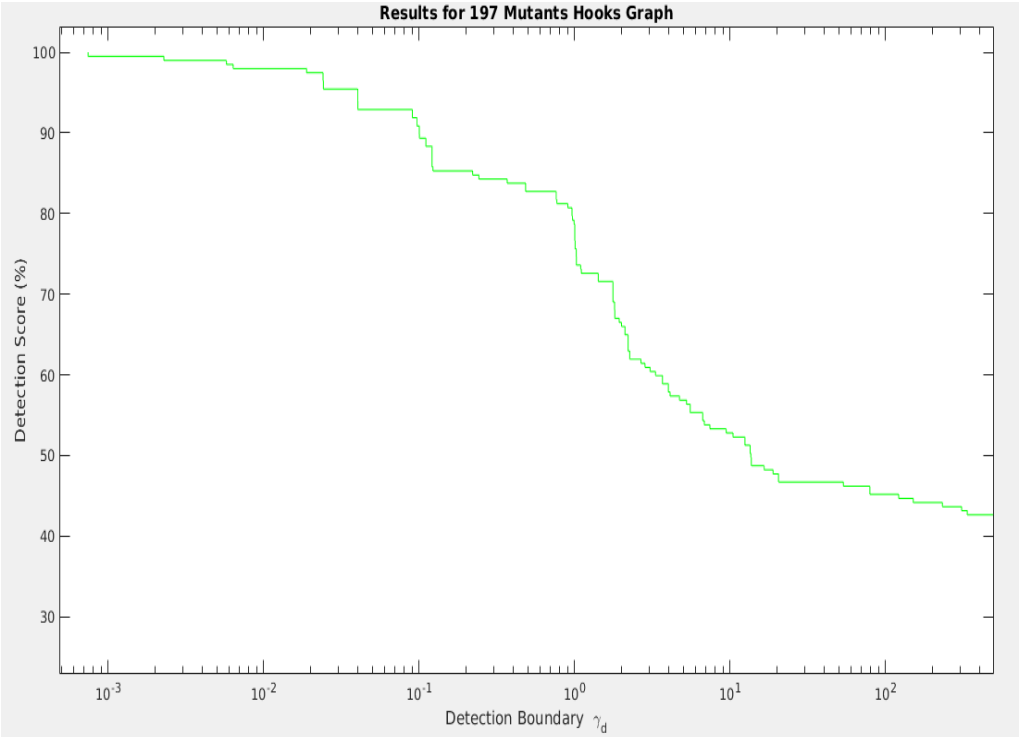
Popperian testing relies more on human effort. A tester must have good domain knowledge and intuition to devise tests using Popperian testing. Also, testers must have a good understanding of the function which is under test. This type of approach takes a tremendous amount of time and energy to analyze and generate new test cases.

4.1.2 Pseudo-Random Testing

In this testing technique, test cases are chosen randomly. Initially, the boundaries are defined for the function under test. Then random values are chosen in-between these boundaries. Random testing is not effective compared to Popperian testing. The random values are chosen in MATLAB using the rand function. This technique is easy to implement and test cases are found quickly. However, the test cases are not that effective in killing mutants.

4.2 Automated Effective Test Suite Generation

The main idea of this thesis is to find test cases that detect mutants at high relative error. Let us look at an example graph to help explain.



Graph 3: Mutant detection graph for sphereFnet function

In Graph 3, the X-axis represents the mutant detection boundary and the Y-axis represents the mutation score for the test suite used. The detection boundary is the relative error at which mutants are detected. The detection score specifies the percentage of mutants killed at a particular point. To generate Graph 3, the test suite TPop was applied to the function sphereFnet. In Graph 3, TPop gets a 100% mutation score when the detection boundary or tolerance value is a little less than 10^{-3}. If scientists know that 10^{-4} is the tolerance value for the sphereFnet function, then test suite TPop will be a good test suite as its detection boundary for 100% detection is greater than 10^{-4}. If the tolerance value is 10^{-2}, then the test suite TPop is not good enough to test the sphereFnet function. Extra test cases must be added to TPop.

Scientific code deals with real values rather than whole numbers, and this results in acknowledged and roundoff errors. Due to the acknowledged errors, scientists find it difficult to predict or find the tolerance value they are working with. If scientists do not know the tolerance value, then they cannot be confident on the test suite being used. To overcome this problem, we propose a new way of finding test cases such that mutants get detected at high relative error.

Our idea is to detect each mutant at the highest possible relative error. Let us look at a simple code example below to elaborate the idea.

Original code:

```
y = x + 10;
```

Mutated line of code:

```
y = x * 10;
```

In this example we will use different test cases and access the relative errors obtained.

When x = 1

Original result = 11

Mutant result = 10

$$\text{relative error} = \frac{|\,mutant\ result - original\ result\,|}{|\,original\ result\,|}$$

$$\text{relative error} = \frac{|\,10 - 11\,|}{|\,11\,|}$$

$$\text{relative error} = \frac{1}{11} = \mathbf{0.0909}$$

When x = 10

Original result = 20

Mutant result = 100

$$\text{relative error} = \frac{|\,100 - 20\,|}{|\,20\,|}$$

$$\text{relative error} = \frac{80}{20} = \mathbf{4}$$

When x = 100

Original result = 110

Mutant result = 1000

$$\text{relative error} = \frac{|\,1000 - 110\,|}{|\,110\,|}$$

$$\text{relative error} = \frac{890}{110} = \mathbf{8.09}$$

From the above example, we can see an increase in the relative error with a change of test case value (x). An increase in relative error can happen when appropriate test cases are chosen. We found that choosing test cases to detect mutants at high relative error can

result in a 100% mutation kill at high relative error. This pushes the graph towards to the right. As most scientists find it difficult to determine a tolerance value, it will always be better to detect mutants at high relative error.

Further in this chapter our proposed way of generating test cases to detect mutants at high relative error is presented.

4.2.1 Manual Generation of Test Cases

Initially, test cases to detect at high relative error were found manually by conducting an experiment. In this experiment, a mutant which is killed at a very low relative error for a function nwtsqrt (Newton square root) is taken into consideration. I tried to find a test case to detect this mutant at the highest possible relative error. The nwtsqrt function takes two inputs and uses Newton's method to find the square root of a number. To find the right test case we need to search the 3D space of the mutant, where the X-axis is the first parameter of the function nwtsqrt, the Y-axis is the second parameter of nwtsqrt, and the Z-axis is the relative error between the mutant and original function (nwtsqrt). A point in 3D space represents the relative error between mutant and original function for particular X and Y input values. If a function has 3 input parameters then we need to search a 4D space to get the test case that detects a mutant at high relative error, as we need an extra dimension for the third input parameter. The code of nwtsqrt and its mutant can be seen below.

The MATLAB code for nwtsqrt and its mutant:

```
%code for nwtsqrt(Newton's square root)
function y = nwtsqrt(x, init)
tol = 1e-10;
y = init;
while abs(y*y - x) > tol
    y = (y + x/y)/2;
end
```

```
%code for nwtsqrt's mutant
function y = nwtsqrt(x, init)
tol = 9e-11; % constant replaced: 1e-10 -> 9e-11
y = init;
while abs(y*y - x) > tol
        y = (y + x/y)/2;
end
```

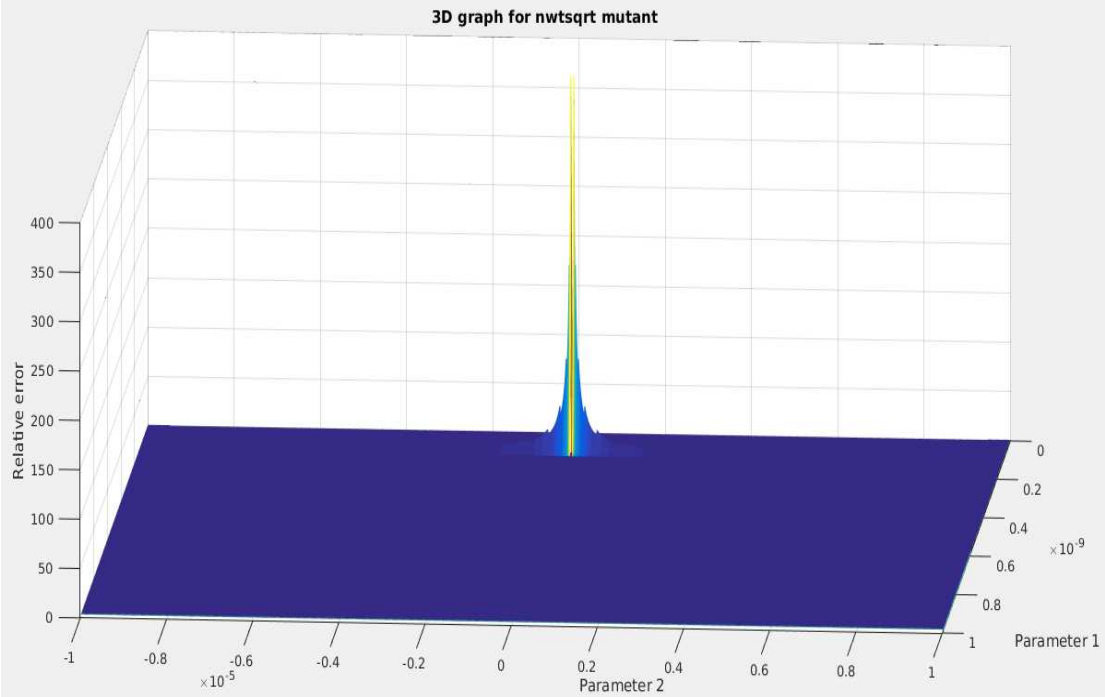
A MATLAB script was written to search the 3D space and to find the best possible test case. This script is shown below.

MATLAB script to generate a 3D graph for nwtsqrt and its mutant

```
%lower bounds of parameters
first_operator_initial = 1e-15; %1e-11
```

```
second_operator_initial = 1e-10; %1e-8
%upper bounds of paramter
first_operator_end = 1e-9; %1e-9
second_operator_end = 1e-5; %1e-5

first_operator_incrementor = first_operator_initial;
second_operator_incrementor = second_operator_initial;

first_loop_count = 1;
actual_results_2d = [];
mutant_results_2d = [];
absolute_error_2d = [];
relative_error_2d = [];

Parameter1 = [];
Parameter2 = [];
relerror = [];
all_loop_counter = 1;
%loop through the first operator
while first_operator_incrementor <= first_operator_end

        second_loop_count = 1;
        %loop through the second operator
        while second_operator_incrementor <= second_operator_end

        %calculate original function value
        actual_results_2d(first_loop_count,second_loop_count)                                   =
nwtsqrt(first_operator_incrementor,second_operator_incrementor);

        %calculate mutant output value
```

```
        mutant_results_2d(first_loop_count,second_loop_count) =
Num6Line7Vers6(first_operator_incrementor,second_operator_incrementor);

        %calculate absolute error b/w original and mutant
        absolute_error_2d(first_loop_count,second_loop_count) =
abs(actual_results_2d(first_loop_count,second_loop_count) -
mutant_results_2d(first_loop_count,second_loop_count));

        %calculate relative error b/w original and mutant
        relative_error_2d(first_loop_count,second_loop_count) =
(absolute_error_2d(first_loop_count,second_loop_count)./abs(actual_results_2d(first_loop_count,second_loop_count))
);

        Parameter1(first_loop_count,second_loop_count) = first_operator_incrementor;
        Parameter2(first_loop_count,second_loop_count) = second_operator_incrementor;
        % store the relative error in 2-D array
        relerror(all_loop_counter,1) = relative_error_2d(first_loop_count,second_loop_count);
        second_operator_incrementor = (second_operator_incrementor + (1e-10 * 2)) ; %1e-9 * 2
        second_loop_count = second_loop_count + 1;

        all_loop_counter = all_loop_counter + 1;
        end

        first_operator_incrementor = (first_operator_incrementor + (1e-11 * 2)) ; %1e-11 * 2
        first_loop_count = first_loop_count + 1;
        second_operator_incrementor = second_operator_initial;
end
%Plot the graph between parameter1, parameter2 and relative error
figure
meshz(Parameter1,Parameter2,relative_error_2d);
```



Graph 4: A 3D graph to find a test case for the nwtsqrt mutant

Graph 4 shows the 3D space comprising the input parameters of the function and relative error. In Graph 4, the X-axis represents the first parameter of the function, the Y-axis represents the second parameter of the function, and the Z-axis represents the relative error. So in order to find the test case we need to find the best parameter combination for which relative error is maximum. If we observe the graph, the maximum relative error peak of 360 (approx.) is found at ($0.2 * 10^{-9}$ (approx.), $0.1 * 10^{-5}$ (approx.)).

Searching the n-dimensional space (n is the number of parameters for the function under test) is very costly as we have to check each and every point in the space. This process of searching the n-dimensional space can be made more efficient by using several optimization algorithms. In this thesis, I used a genetic algorithm to optimize the search process.

4.2.2 Genetic Algorithm

A genetic algorithm is good at efficiently searching a huge search space. It looks for an optimal combination of parameters for which maximum fitness is obtained. Its working is based on Darwin's principle of natural selection. Instead of searching all the members in a search space, a genetic algorithm initially selects a certain number of distributed random members and finds the fitness value of each member. It then finds the best member among the random members and checks more members near to the member found. It keeps on searching until it finds a good solution. The working of a genetic algorithm is explained in the steps below:

1. An initial random population is created, distributed across the search space.
2. A fitness value is computed for each member in the population.
3. Parents are selected based on fitness scores. Members with high fitness scores get selected as parents.
4. Members with the highest fitness scores are selected as elite members and are passed on to the next generation.
5. Children are produced from the parents using two processes: mutation and crossover. In mutation, a child is produced by making random changes to a single parent. In crossover, a child is produced by combining the values of a pair of parents.
6. The next generation population will consist of the elite members and the children produced.

7. The whole process repeats until a stopping criterion is met. A generation limit is an example of a stopping criterion where the process stops when a certain number of generations are reached. Another example is a fitness limit. If the fitness value is greater than or equal to the fitness limit then the process stops.

In this chapter I will discuss how this mutation testing process is applied to automatic test suite generation.

4.2.3 Automated Test Suite Generation Using Genetic Algorithm

A manual, brute force search of N-dimensional space is computationally costly. So to explore the search space effectively and to reduce the computational cost, a genetic algorithm is used. The main process for generating an effective automated test suite is explained in steps below:

1. To generate a test suite for a particular function, first generate mutants for the function using pymute.
2. For each mutant, use a genetic algorithm to search the space to find a test case that gives the highest possible relative error.
3. Add the test case found by the genetic algorithm to the test suite. Repeat step 2 and step 3 until all the mutants are processed.

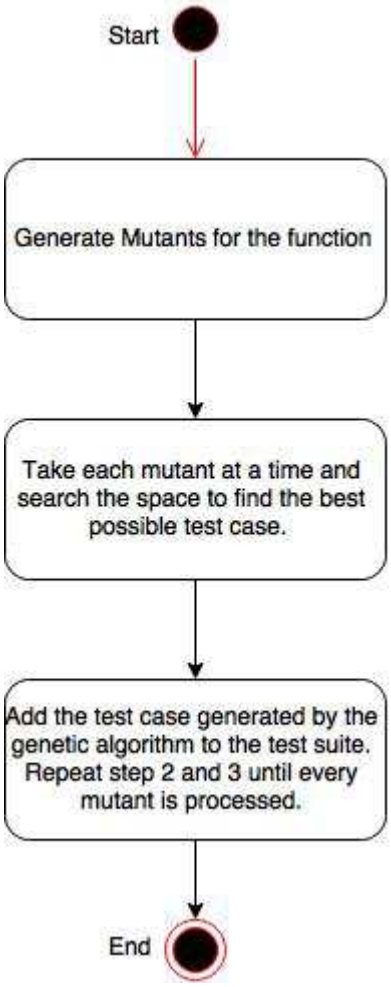


Figure 4: Test suite generation process

Now let us look a little closer at the test suite generation process. To execute this process three modules or files were programmed. These three modules are Test_suite_generation_script, Genetic_solver, and Fitness_function. The code in each module is shown and the working is explained.

4.2.3.1 Test_suite_generation_script

The main work for this module is to create mutants and call Genetic_solver. This

function takes the function name and number of input parameters of the function and gives back an auto generated test suite. Initially the loop monitoring variables are set (see Chapter 3) and loop monitoring performed. Next the Python module pymute is called from this script and mutants are created and stored in a folder. For each mutant a call to Genetic_solver is made to get a test case that detects the mutant at the highest possible relative error. All the test cases are stored in a format such it can be used in MATmute to get the detection graph. The MATLAB code in Test_suite_generation_script is shown below.

Test_suite_generation_script:

```
function [genetic_test_set] = Test_suite_generation_script(fn_name,no_variables)
        loop_timing_factor = 100;
        fail_on_bad_loop = true;

        % used for loop monitoring
        global MUTE_ticks MUTE_ticklimit MUTE_testcnt
        global MutantName

        % If function's mutant directory exists then erase it's contents so that
        % the new mutants will not be mixed with old mutants.
        if isdir([pwd '/' fn_name '_mutes'])
        rmdir([fn_name '_mutes'], 's')
        end

        initdir = pwd; % store the current directory
        initpath = path; % store current path so it can be restored
        initwarning = warning('query', 'all');
```

```
% Generate the call string which calls the mutator.
callstring = ['python -m matmute ' fn_name];

% If fail_on_bad_loop is true then append "--inferr" flag to call string.
if fail_on_bad_loop
callstring = [callstring ' --inferr'];
end

% Call the mutator using the callstring.
[status output] = system(callstring);

% Check if the mutator reported any errors.
if status == 0
disp('Creation of mutants complete.')
else
disp(output);
error('Mutator encountered a problem when creating the mutants.  See error message above.')
end

% Parse the mutant IDs from the mutator's output string.
mute_ids = regexp(deblank(output), '\n', 'split');

% If empty string is returned as the mutant IDs then quit.
if isempty(mute_ids)
error('No mutants created.  Perhaps you should try different mutation operators?')
else
fprintf('Tests will be executed on %i mutants.\n', length(mute_ids))
end

% Add the current directory to the path (so that any functions in this
```

```
% directory are still accessible).
path(initdir, path);

% Change to the directory where the mutants are stored.
cd([fn_name '_mutes']);

% Use try-catch to ensure that MATLAB returns to the original directory.
try

% MUTE_testcnt counts the number of tests that have been executed on
% each mutant.  It is used by the mutated files to determine which
% ticklimit corresponds to each test, and must be reset every time a new
% mutant is being analysed.
MUTE_testcnt = 0;

% Execute tests on the unmutated function and compare the results with
% the results from original function.

% Use results from the instrumented version to set loop iteration limits.
% (+1 ensures that ticklimit is nonzero.)
MUTE_ticklimit = (MUTE_ticks+1)*loop_timing_factor;

%genetic_test_set = zeros(length(mute_ids), 1);
count = 1;
%To get initial population for genetic algorithm
%initialPop = Convert_cellarray_to_array();
% Loop through all the mutants.
for m = 1:length(mute_ids)

MUTE_testcnt = 0; % as above, MUTE_testcnt must be reset
```

```
% Turn off warnings (except desired mutant warnings) while
% running mutants.
warning('off', 'all')
warning('on', 'mutant:TickLimitExpired')
warning('on', 'get_error:RelErrDivByZero')
warning('on', 'get_error:NaN')
warning('on', 'get_error:Inf')
try
%create object for genetic solver and call the function
%genetic test find.
geneticTestObject = Genetic_solver_SphereFnet;
geneticTestObject.mutantname = mute_ids{m};
Genetic_test_case =geneticTestObject.geneticTestFind(no_variables,mute_ids{m});

%save the test case in a format such that it can be used in
%MATmute
genetic_test_set.sphereFnet{count,1} = Genetic_test_case;
count = count + 1;

m
catch ME
s = ME.stack(1);

end
% Turn warnings back on.
warning(initwarning)

end
catch ME
```

```
        % Return to main directory, reset variables and pass error along.
        cd(initdir);
        path(initpath);
        warning(initwarning)
        clear global MUTE_ticks MUTE_ticklimit MUTE_testcnt
        rethrow(ME)

        end

        % Change back to the original directory and remove addition to the path.
        cd(initdir);
        path(initpath);

        % Reset global variables.
        clear global MUTE_ticks MUTE_ticklimit MUTE_testcnt

        % Display success message.
        disp('Tests finished executing.')

end
```

4.2.3.2 Genetic_solver

In the Genetic_solver module, a genetic algorithm is called with customized options and a test case is returned. In the Genetic_solver class, the main implementation of the logic is done in the geneticTestFind function. This function takes the number of input parameters and the mutant name as input, and gives back the test case found by the genetic algorithm. MATLAB has a predefined genetic algorithm functionality in the Global Optimization Toolbox. This predefined function is used in this thesis. This genetic

algorithm has a vast number of options, and the details about the options can be found at the link http://www.mathworks.com/help/gads/genetic-algorithm-options.html. For the genetic algorithm, the lower and upper bounds of each parameter, the number of parameters, and fitness function are given as input. The genetic algorithm options play a crucial role in the performance. Decisions such as turning off the crossover fraction and choosing a population size of 500 are made by experimentation. It was observed that crossover was not useful while generating test cases, so this option was turned off. The genetic algorithm is multi-threaded and works faster on machines with more cores. All the options used can be seen in the code below. Finally, the genetic algorithm returns a test case that detects the mutant at the highest possible relative error, or at least very close to the highest possible relative error.

Genetic_solver:

```
classdef Genetic_solver_SphereFnet
        properties
        numberOfVariables
        mutantname
        end
        methods
        function testcase = geneticTestFind(obj,numberofVariables,currentmute)


        for loop = 1 : 1


%Fitness function
        ObjectiveFunction = @(x)simple_fitness_SphereFnet(x,currentmute);
% Variables number must be given to GA
        nvars = numberofVariables;
```

```
% Lower bound of input parameters
        LB = [eps 1 1 1 1 eps];
% Upper bound of input parameters
        UB = [10^7 1000 1000 1000 1000 10^6];


% FitnessLimit, TolFun and StallGenLimit are stopping criteria
% UseParallel is multi threading option
% generations is a stopping criteria
% population size specifies the number of members
% Crossover fraction is set to 0 because it is not useful at all for us.
opts = gaoptimset(' FitnessLimit ',-1,' StallGenLimit ', 100, ‘ TolFun ’,0, 'UseParallel','always','Generations',200,'
CrossoverFraction ',0, ' PopulationSize ',500);
        %Call GA with the options defined.
        [x,Fval] = ga(ObjectiveFunction,nvars,[],[],[],[],LB,UB,[],opts);
        Fval
        all_tests(loop,:) = [x,Fval];


        end
        % retrieve the test case and return it
[relative_err,index] = min(all_tests(:,numberofVariables + 1));
        best_test = all_tests(index,:);
        testcase = {best_test(1),best_test(2),best_test(3),best_test(4),best_test(5),best_test(6)};


        end
        end
end
```

4.2.3.3 Fitness_function

A genetic algorithm requires a measure to differentiate between good and bad members of the population, and that measure is provided by a fitness function. For every member

in the population, the fitness function is called, and the value returned by the fitness function is assigned to each member as its fitness value. In this thesis each member in the population is a test case and its fitness value is the relative error. Fitness_function takes a member of the population and mutant name as input, and gives the relative error as output. In the Fitness_function module the actual result and mutant output are calculated and the relative error is found. The fitness_function code is easy to understand and can be seen below.

Fitness_function:

```
function relativeError = Fitness_function_SphereFnet(x,mutant)

        global MUTE_testcnt;
        %Initialize variables
        actualValue = cell(1, 1);
        muteValue = cell(1, 1);
        %Calculate actual value
        actualValue{1} = sphereFnet(x(:,1),x(:,2),x(:,3),x(:,4),x(:,5),x(:,6));
        try
        handle = str2func(mutant);
        %Calculate mutant output
        muteValue{1} = feval(handle,x(:,1),x(:,2),x(:,3),x(:,4),x(:,5),x(:,6));
        catch ME
        s = ME.stack(1);
        muteValue{1} = sprintf('_MUTE_err caught originating from line %d of "%s":\n%s', s.line, s.name,
ME.message);

        end
        MUTE_testcnt = 0 ;
```

```
%find relative error
relativeError = max(get_error(actualValue,muteValue));
% Genetic algorithm in matlab tries to find the smallest value,
% But we need maximum relative error. So the sign is changed.
relativeError = -relativeError;
end
```

4.2.3.4 Mutation and Initial Population Creation

The genetic algorithm works fine for those functions with numerical input parameters (such as double, int, and float). But the MATLAB predefined genetic algorithm won't work for functions with input parameters other than numbers. For functions with input parameters such as array and string, the tester must develop his own functions to create an initial population, and to perform crossover and mutation on the members. These developed functions must be given as input to the genetic algorithm to generate test cases. While writing initial population creation functions, my aim was to uniformly distribute the test cases in n-dimensional space. Developing a mutation function depends on the function that is under test. Initial population creation and mutation function code for a binary search function can be seen below.

Population_Creation:

```
function pop = Population_Creation(NVARS,FitnessFcn,options)
%CREATE_PERMUTATIONS Creates a population of permutations.
%   POP = CREATE_PERMUTATION(NVARS,FITNESSFCN,OPTIONS) creates a population
%  of permutations POP each with a length of NVARS.
```

```
%
%   The arguments to the function are
%         NVARS: Number of variables
%         FITNESSFCN: Fitness function
%         OPTIONS: Options structure used by the GA

totalPopulationSize = sum(options.PopulationSize);
n = NVARS;
pop = cell(totalPopulationSize,2);
arraySizeIncrementor = 1;
ValueIncrementor = 1;
for i = 1:totalPopulationSize
        if arraySizeIncrementor > 25
        arraySizeIncrementor = 1;
        end
        if ValueIncrementor > 10
        ValueIncrementor = 1;
        end
        pop{i} = sort(randi([-10.^ValueIncrementor 10.^ValueIncrementor],1, arraySizeIncrementor));
        pop{i,2} = pop{i}(randi([1 size(pop{i},2)]));
        arraySizeIncrementor = arraySizeIncrementor + 1;
        ValueIncrementor = ValueIncrementor + 1;
End
```

Mutation_binSearch:

```
function mutationChildren = Mutation_binSearch(parents ,options,NVARS, ...
        FitnessFcn, state, thisScore,thisPopulation,mutationRate)
%   MUTATE_PERMUTATION Custom mutation function for traveling salesman.
```

```
%   MUTATIONCHILDREN = MUTATE_PERMUTATION(PARENTS,OPTIONS,NVARS, ...
%   FITNESSFCN,STATE,THISSCORE,THISPOPULATION,MUTATIONRATE) mutate the
%   PARENTS to produce mutated children MUTATIONCHILDREN.
%
%   The arguments to the function are
%       PARENTS: Parents chosen by the selection function
%       OPTIONS: Options structure created from GAOPTIMSET
%       NVARS: Number of variables
%       FITNESSFCN: Fitness function
%       STATE: State structure used by the GA solver
%       THISSCORE: Vector of scores of the current population
%       THISPOPULATION: Matrix of individuals in the current population
% In this function extra element is added to the list,
mutationChildren = cell(length(parents),2);
for i=1:length(parents)
        getArray = thisPopulation{parents(i)};
        arraySize = size(getArray,2);
        MuteType = randi([1 2]);
        if arraySize < 25
        getArray(arraySize + 1) = randi([-10.^10 10.^10]);
        end
        mutationChildren{i} = sort(getArray);
        mutationChildren{i,2} = getArray(randi([1 size(getArray,2)]));
end
```

4.2.4 Independent Oracle Creation and Unit Testing

An oracle is usually unavailable to scientists as they try to implement new ideas. To mitigate this problem to some extent, I propose an idea to replicate the MATLAB code in

R or Java. These replications can be considered as oracles. There are two important uses for this type of oracle creation. First, in rare cases inbuilt MATLAB functions used in the code may exhibit round off or acknowledged errors. Second, if any MATLAB code is changed or updated for any reason, then the MATLAB code may be unit tested using these oracles to check whether the modifications or updates have changed the original behavior.

For all the 8 functions mentioned in Chapter 5 and for 1 function from the LUCY package, oracles are created using either R or Java. The 8 functions are unit tested using the oracles created. For example, sphereFnet is one of the 8 functions and is shown below along with its oracles.

MATLAB Code for sphereFnet:

```
function out = sphereFnet(v, m, d, g, p, u)

        Re = p*v*d/u;
        cd = calcCd(Re);
        out = m*g - cd*0.5*p*v^2*pi*d^2/4;

function cd = calcCd(Re)
        data = load('sphereCd.dat');
        if Re < 0
        error('Negative Re value encountered')
        elseif Re <= 2e4
        cd = 24/Re + 6/(1+sqrt(Re)) + 0.4;
        elseif Re <= 3.99e6
        cd = interp1(data(:,1), data(:,2), Re);
```

```
        else
        cd = 0.1810;
        end
```

Oracle for sphereFnet using R:

```
require("pracma",quietly=TRUE);
sphereFnet <- function(v, m, d, g, p, u) {
  Re <- p*v*d/u;
  cd <- calcCd(Re);
  #round(cd, 16)
  out <- m*g - cd*0.5*p*v^2*pi*d^2/4;
  output <- format(out,digits = 22)
  cat(output)
}
calcCd <- function(Re) {
  x <- c(20000, 38200, 73000, 144000, 220000, 258000, 293000, 319000, 341000, 350000, 365000, 384000,  417000,
462000, 548000, 743000, 1190000, 2050000, 3220000, 3990000)
  y <- c(0.4430, 0.4900, 0.5030, 0.5150,  0.5080, 0.4940, 0.4740, 0.4360, 0.3740, 0.2580, 0.1260, 0.0836, 0.0692,
0.0655, 0.0726, 0.0888, 0.1230, 0.1570, 0.1740, 0.1810)
  if (Re < 0 && !(is.nan(Re))) {
        error('Negative Re value encountered')
  }
  else if (Re <= 2e4 && !(is.nan(Re))) {
        cd <- 24/Re + 6/(1+sqrt(Re)) + 0.4;
  }
  else if (Re <= 3.99e6 && !(is.nan(Re))) {
        cd <- interp1(x, y, Re, method = "linear");
  }
```

```
 else
        cd <- 0.1810;
}
```

Oracle for sphereFnet using Java Apache Commons:

```
import org.apache.commons.math3.analysis.UnivariateFunction;
import org.apache.commons.math3.analysis.interpolation.LinearInterpolator;
import org.apache.commons.math3.analysis.interpolation.UnivariateInterpolator;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.lang.Math;
import java.math.BigDecimal;
public class sphereFnet {

   double[] x,y;
   public void getvalues(){
   x= new double[20];

   x[0] = 20000;
   x[1] = 38200;
   x[2] = 73000;
   x[3] =144000;
   x[4] =220000;
   x[5] =258000;
   x[6] =293000;
   x[7] =319000;
   x[8] =341000;
   x[9] =350000;
```

```
x[10] =365000;

x[11] =384000;

x[12] =417000;

x[13] =462000;

x[14] =548000;

x[15] =743000;

x[16] =1190000;

x[17] =2050000;

x[18] =3220000;

x[19] =3990000;



 y = new double[20];


y[0] = 0.4430;

y[1] = 0.4900;

y[2] = 0.5030;

y[3] = 0.5150;

y[4] = 0.5080;

y[5] = 0.4940;

y[6] = 0.4740;

y[7] = 0.4360;

y[8] = 0.3740;

y[9] = 0.2580;

y[10] =0.1260;

y[11] =0.0836;

y[12] =0.0692;

y[13] =0.0655;

y[14] =0.0726;

y[15] =0.0888;
```

```
y[16] =0.1230;
y[17] =0.1570;
y[18] =0.1740;
y[19] =0.1810;

}
sphereFnet(double v, double m,double d,double g,double p,double u) throws FileNotFoundException, IOException
{
        double Re, cd, out,cr ;
        System.getProperty("sun.arch.data.model");
        Re = p*v*d/u;
       cd = calcCD(Re);
       out = m*g - cd * 0.5 * p * Math.pow(v,2) * 3.141592653589793 * Math.pow(d, 2)/4;
       System.out.println(String.format("%.22f",out));
       //System.out.println(BigDecimal.valueOf(out));
}

private double calcCD(double Re) throws FileNotFoundException, IOException{
        getvalues();
        double twentythousand = 20000;
        double thirtyninelakhs = 3990000;
        double cd = 0;
        if (Re < 0)
                System.out.println("Negative Re value encountered");
        else if (Re <= twentythousand){
                cd = 24/Re + 6/(1+Math.sqrt(Re)) + 0.4; //constant replaced: 24 -> 23
        }
        else if (Re <= thirtyninelakhs){
                UnivariateInterpolator i=new LinearInterpolator();
                UnivariateFunction uf=i.interpolate(x,y);
```

```
                cd = uf.value(Re);
        }
        else
                cd = 0.1810;

        return cd;

    }
    public static void main(String[] args) throws FileNotFoundException, IOException {
        // TODO Auto-generated method stub
        double v,m,d,g,p,u;
        v = Double.parseDouble(args[0]) ; m = Double.parseDouble(args[1]); d = Double.parseDouble(args[2]);
        g = Double.parseDouble(args[3]); p = Double.parseDouble(args[4]); u = Double.parseDouble(args[5]);
        sphereFnet sp = new sphereFnet(v,m,d,g,p,u);
    }
}
```

4.2.4.1 Implementation of Unit Testing

Unit testing is done on all 8 functions and on 1 function from the LUCY package. The replications in R or Java act as oracles as said before, and the set of test cases developed using our proposed technique are taken as the test suite for unit testing. Unit testing is performed using the MATLAB unit testing framework. For every function a Unittesting script is created such that, when string ‘java’ is given by the user then it unit tests the MATLAB function with the Java oracle. The Unittesting script uses the R oracle when string ‘R’ is given as input.

To explain the process, let us consider the Unittesting script of the sphereFnet function

(SphereFnetUnittesting). If the SphereFnetUnittesting script is called by the user with input string as ‘java’ then each test case is sent one by one to sphereFnetjavaapache script. The sphereFnetjavaapache script will call the java oracle which is a jar file and fetches the result. The results from the oracle and MATLAB function are compared and the unit test result (pass or fail), is sent back to SphereFnetUnittesting. This process repeats for all the test cases and the count of total unit tests passed is printed. The code used for unit testing can be seen below.

SphereFnetUnittesting Script:

```
function SphereFnetUnittesting(platform)

    load('Genetic_sphereFnet_main.mat');
    relRcount = 0;
    for row = 1 : size(TPop.sphereFnet,1)
    spherev = TPop.sphereFnet{row,1}{1};
    spherem = TPop.sphereFnet{row,1}{2};
    sphered = TPop.sphereFnet{row,1}{3};
    sphereg = TPop.sphereFnet{row,1}{4};
    spherep = TPop.sphereFnet{row,1}{5};
    sphereu = TPop.sphereFnet{row,1}{6};

    disp('-----------------------------------------------------')
    disp('-----------------------------------------------------');
    fprintf('evaluating SphereFnet testcase no %d',row);
    disp(' ');

    if platform == 'R'
    fprintf('evaluating R SphereFnet testcase no %d',row);
```

```
disp(' ');
test1 = sphereFnetRscript;
test1.spherev = spherev;
test1.spherem = spherem;
test1.sphered = sphered;
test1.sphereg = sphereg;
test1.spherep = spherep;
test1.sphereu = sphereu;
testresult = run(test1);
end
if platform == 'java'
fprintf('evaluating java SphereFnet testcase no %d',row);
disp(' ');
test1 = sphereFnetjavaapache;
test1.spherev = spherev;
test1.spherem = spherem;
test1.sphered = sphered;
test1.sphereg = sphereg;
test1.spherep = spherep;
test1.sphereu = sphereu;
testresult = run(test1);
end
if testresult.Passed == 1

relRcount = relRcount + 1;

fprintf('testcase no %d is passed ',row);
disp(' ');
disp('-----------------------------------------------------');
disp('----------------------END-------------------------')
```

```
        disp(' ');
        disp(' ');
        end
        end
        fprintf('No of unit tests passed %d\n',relRcount);
end
```

Unit testing MATLAB code using Java oracle:

```
classdef sphereFnetjavaapache < matlab.unittest.TestCase
        %UNTITLED2 Summary of this class goes here
        %   Detailed explanation goes here

        properties
        spherev
        spherem
        sphered
        sphereg
        spherep
        sphereu

        end


        methods (Test)
        function testsphereFnet(testcase)

        v = testcase.spherev;
        m = testcase.spherem;
        d = testcase.sphered;
        g = testcase.sphereg;
```

```
p = testcase.spherep;
u = testcase.sphereu;

initdir = pwd; % store the current director
initpath = path; % store current path so it can be restored
targetdir = '/home/subhash/Desktop';
javaargument = 'java -jar sphereFnet.jar ';
path(initdir, path);

actualSolution = sphereFnet(v, m, d, g, p, u);
sprintf('actual solution is %16.f',actualSolution)

cd(targetdir);
callstring = [javaargument num2str(v,64) ' ' num2str(m,64) ' ' num2str(d,64) ' ' num2str(g,64) ' '   num2str(p,64)
' ' num2str(u,64)];
[status,cmdout] = unix(callstring);
resultfromjava = str2double(cmdout);
sprintf('Java result is %16.f',resultfromjava)


tempjavaexpectresult = (sprintf('%.4f',resultfromjava));
javaexpectresult = str2double(tempjavaexpectresult);
tempactualresult = (sprintf('%.4f',actualSolution));
actualresult = str2double(tempactualresult);

cd(initdir);
path(initpath);

testcase.assertEqual(actualSolution,resultfromjava,'RelTol',10^-10);
```

```
        end
        end

end
```

Unit testing MATLAB code using R oracle:

```
classdef sphereFnetRscript < matlab.unittest.TestCase
        %SPHEREFNETRSCRIPT Summary of this class goes here
        %   Detailed explanation goes here

        properties
        spherev
        spherem
        sphered
        sphereg
        spherep
        sphereu

        end

        methods (Test)
        function testsphereFnet(testcase)
        v = testcase.spherev;
        m = testcase.spherem;
        d = testcase.sphered;
        g = testcase.sphereg;
        p = testcase.spherep;
        u = testcase.sphereu;
        initdir = pwd; % store the current directory
```

```
        initpath = path; % store current path so it can be restored
        targetdir = '/home/subhash/Desktop/Thesis_Matmute/Rscripts';
        Rargument = 'Rscript  sphereFnetR.R ';
        path(initdir, path);
        actualSolution = sphereFnet(v, m, d, g, p, u);
        sprintf('actual result is %16.f',actualSolution)
        cd(targetdir);
        callstring = [Rargument num2str(v,64) ' ' num2str(m,64) ' ' num2str(d,64) ' ' num2str(g,64) ' ' num2str(p,64) ' '
num2str(u,64)]
        [status,cmdout] = unix(callstring);
        resultfromR = str2double(cmdout);
        sprintf('R result is %16.f',resultfromR)
        Rexpectresult = str2double(sprintf('%.4f',resultfromR));
        actualresult = str2double(sprintf('%.4f',actualSolution));
        cd(initdir);
        path(initpath);
        testcase.assertEqual(actualSolution,resultfromR,'RelTol',10^-10);
        end
        end
    end
```

5 Results

To evaluate the automated test generation process, automated test suites were created for the 8 functions used by Hook[16] in Mutation Sensitivity Testing. Hook took these 8 functions from an introductory scientific computing course. Hook created popperian test cases (TPop), random test cases (Trnd), and also a combination of both popperian and random test cases (Tcmb). Hook manually analysed each function and came up with good test suites. The (Tcmb) test suite gave good results for the 8 functions in Hook's thesis. If we compare the automated test suites (Tauto) with Hook's test suites, then we can evaluate the quality of the automated test suites. Now let us see the results for each of the function.

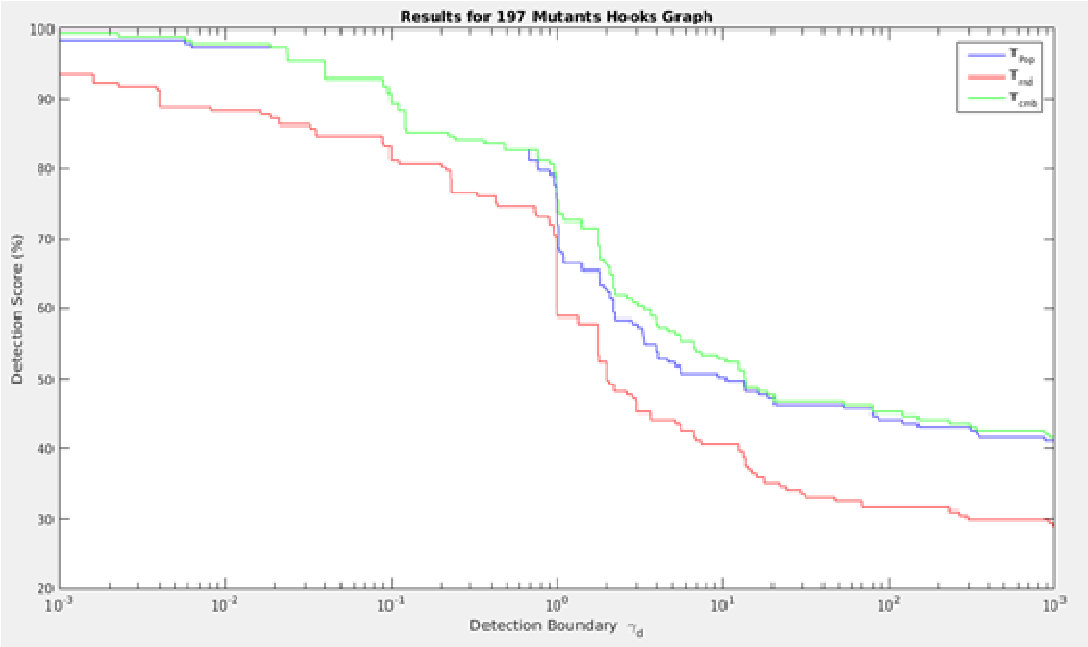
sphereFnet

This function takes 6 inputs. They are velocity of sphere, mass of sphere, diameter of sphere, gravity, density of fluid, and viscosity of fluid. Using these inputs, sphereFnet computes the net force on the sphere when it is falling into a liquid.

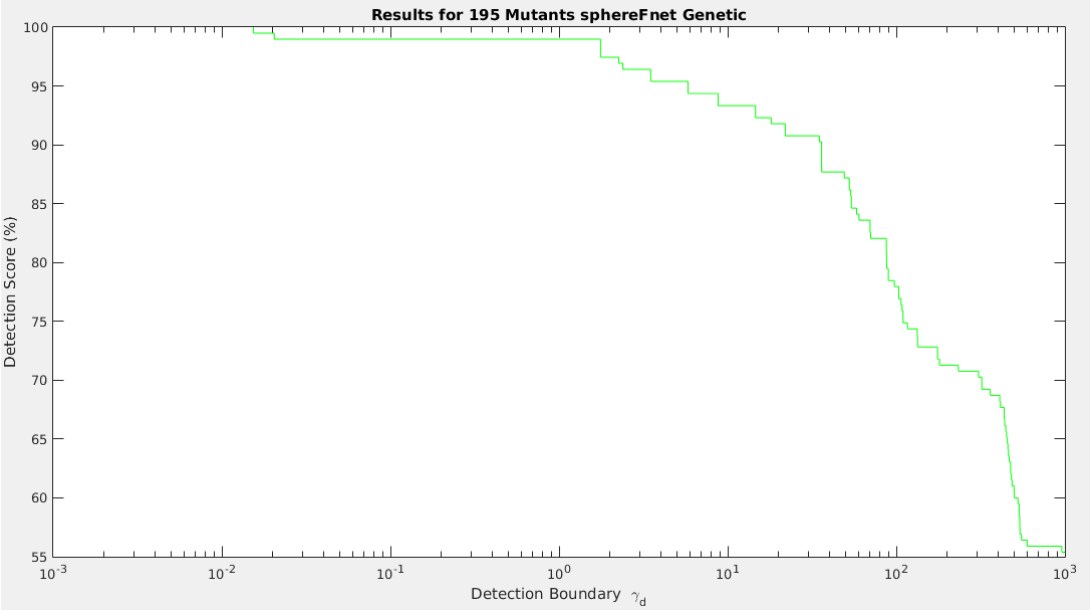
The sphereFnet detection graph for Hook's test suite can be seen in Graph 5 and the detection obtained from the auto generated test suite can be seen in Graph 6. By carefully comparing the graphs the following observations are made:

1. The graph moved to the right for Tauto test suite, this indicates that mutants are detected at high relative error. Also 100 % mutants are detected at high relative error when compared to Hook's graph.

2. Hook's test suites discovered 197 mutants out of 217 and the rest of 20 mutants are either deemed equivalent or terminal failure mutants, whereas Tauto test suite detected 195 out of 217. Fewer mutant detections for Tauto only happened for the sphereFnet function and in the rest of the functions, mutation detections are more. The reason for this might be that Hook added manually a test case after examining a difficult to detect mutant.



Graph 5: sphereFnet detection graph generated using Hook's test suite



Graph 6: sphereFnet detection graph generated using auto generated test suite

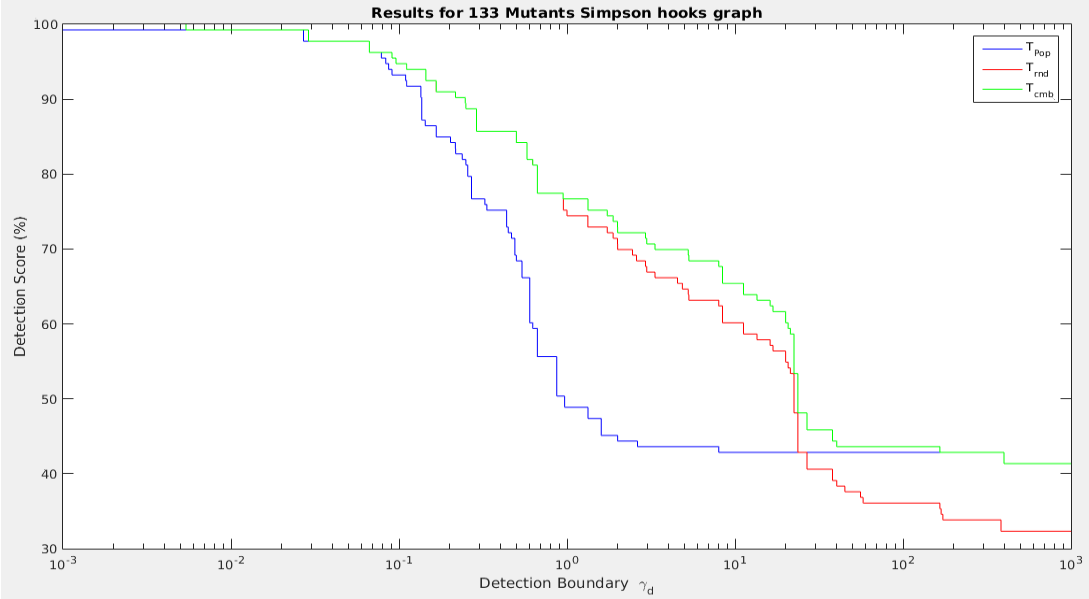
simpson

This function takes 4 inputs: a mathematical function, upper bound of integration, lower bound of integration, and the number of panels for integration. Using these inputs simpson integrates the mathematical function and returns an approximate value.

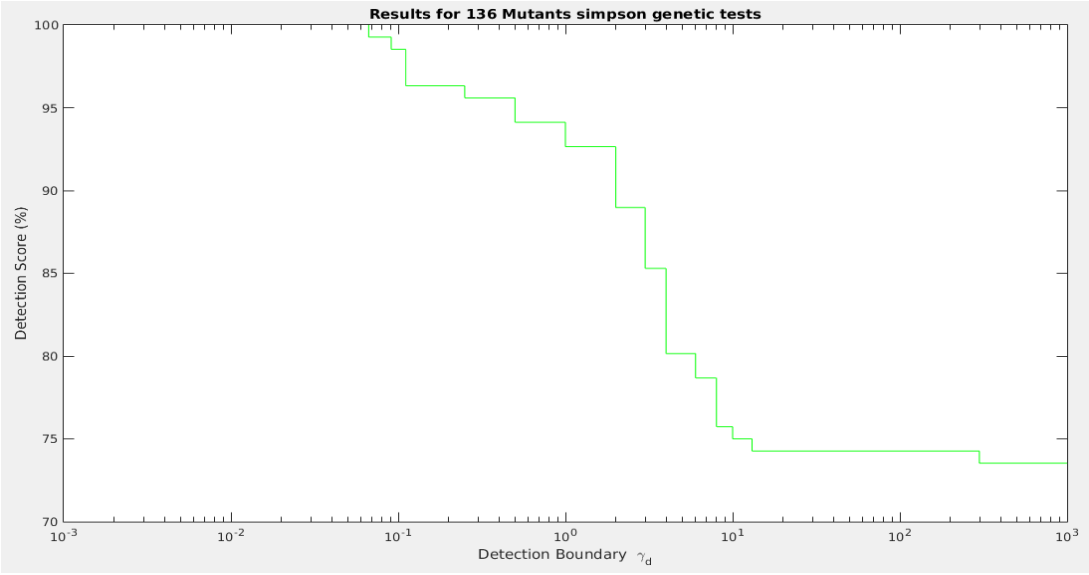
The simpson detection graph for Hook's test suite can be seen in Graph 7 and the detection obtained from the auto generated test suite can be seen in Graph 8. By carefully comparing the graphs the following observations are made:

1. The detection score scale for the graphs are different. The detection scale of Tauto graph is very high when compared to Hook's graph. It can also be clearly observed that mutants got detected at high relative error by Tauto. The 100% mutants detection point is also moved to the right in the Tauto graph.

2. Hook's test suites discovered 133 mutants out of 181. The Tauto test suite detected 136 mutants out of 181. The 3 extra mutants detected by Tauto are considered as equivalent or terminal failure mutants by Hook's test suites.



Graph 7: simpson detection graph generated using Hook's test suite



Graph 8: simpson detection graph generated using the auto generated test suite

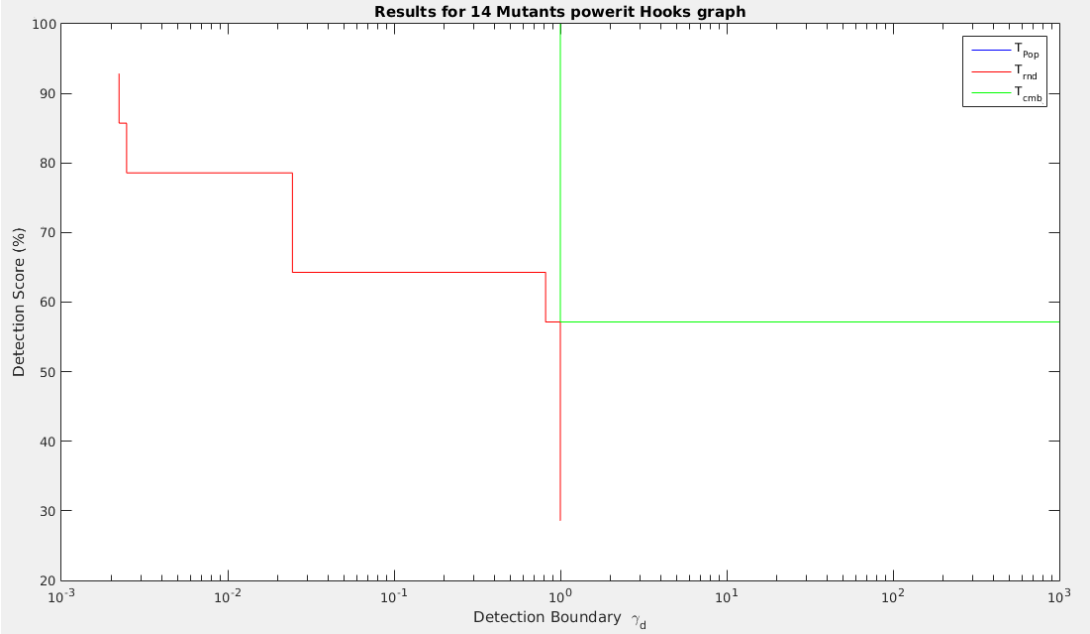
powerit

This function takes 2 inputs: an input matrix and the number of iterations to conduct. Using these inputs, powerit returns an approximate value of the largest eigenvalue of the input matrix.

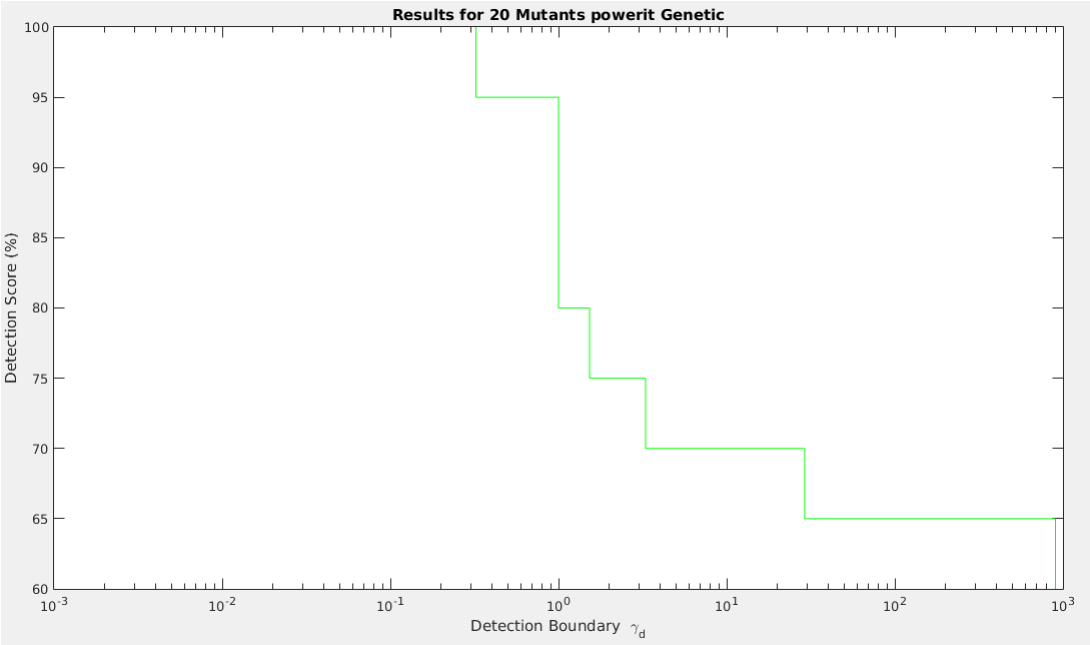
The powerit detection graph for Hook's test suite can be seen in Graph 9 and the detection obtained from the auto generated test suite can be seen in Graph 10. By carefully comparing the graphs the following observations are made:

1. The detection score scale for the graphs are different. The detection scale of the Tauto graph is very high when compared to Hook's graph. The 100% mutation detection is at a lower relative error for Tauto when compared to Hook's test suites. This is because more mutants are detected by Tauto (20) than Hook's test suites (14). Apart from the 100% mutants detection point, the rest of the graph moved towards to the right for Tauto which is a good sign.

2. Hook's test suites discovered 14 mutants out of 32. The Tauto test suite detected 20 mutants out of 32. The 6 extra mutants detected by Tauto are considered as equivalent or terminal failure mutants by Hook's test suites.



Graph 9: powerit detection graph generated using Hook's test suite



Graph 10: powerit detection graph generated using auto generated test suite

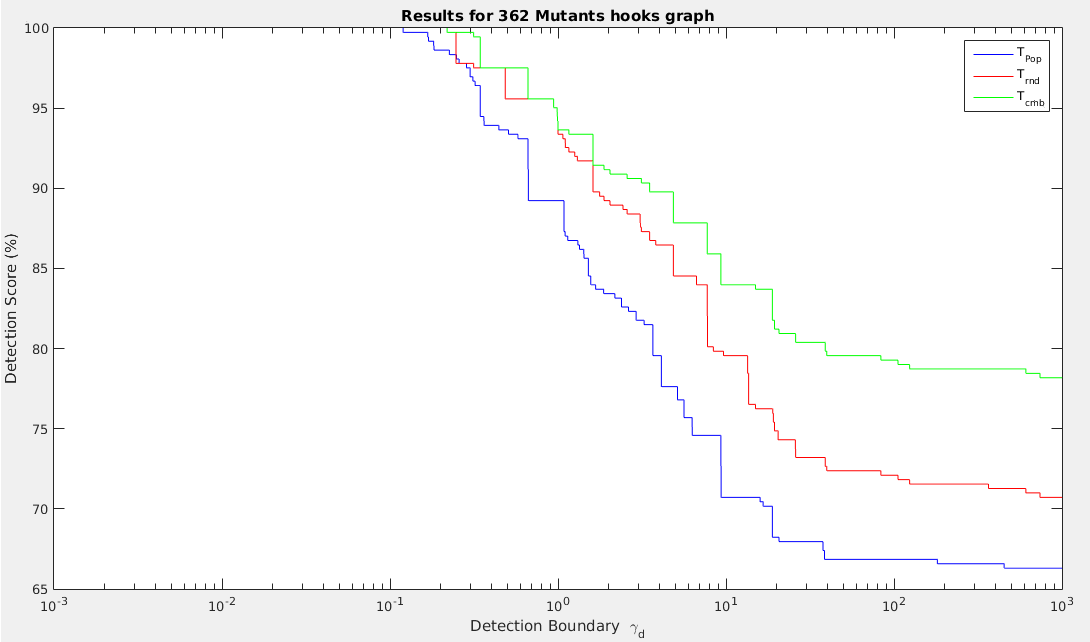
odeRK4

This function takes 4 inputs. They are a mathematical function, stopping value of the variable, step size for the variable, and initial condition of the dependant variable. The odeRK4 function returns a vector containing a numerical solution at each step of the independent variable.

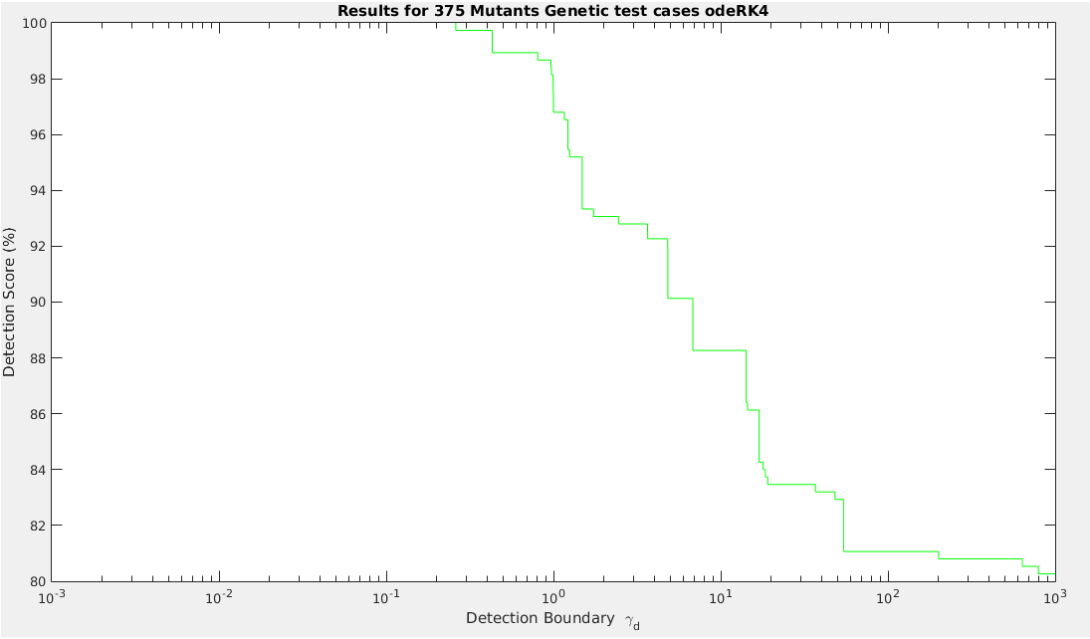
The odeRK4 detection graph for Hook's test suite can be seen in Graph 11 and detection obtained from the auto generated test suite can be seen in Graph 12. By carefully comparing the graphs the following observations are made:

1. The detection score scale for the graphs are different. The detection scale of the Tauto graph is very high when compared to Hook's graph. It can also be clearly observed that mutants got detected at high relative error for the Tauto test suite. The 100% mutants detection point is also moved to the right in the Tauto graph.

2. Hook's test suites discovered 362 mutants out of 424. The Tauto test suite detected 375 mutants out of 424. The 13 extra mutants detected by Tauto are considered as equivalent or terminal failure mutants by Hook's test suites. The rise in the number of mutants detected is a good indication of the increase in quality of the test cases.



Graph 11: odeRK4 detection graph generated using Hook's test suite



Graph 12: odeRK4 detection graph generated using auto generated test suite

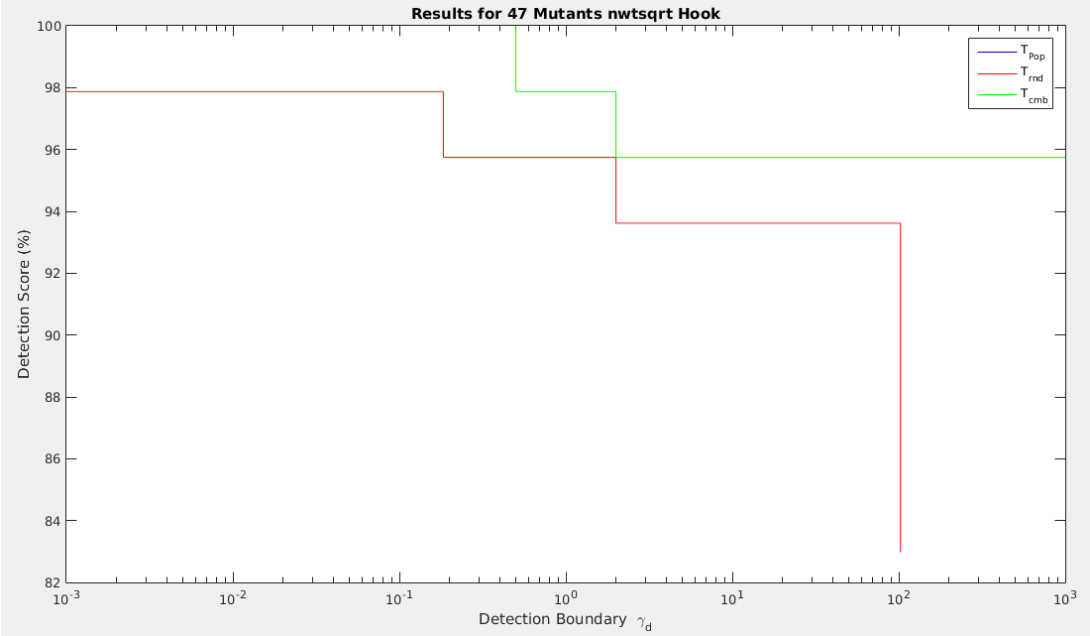
nwtsqrt

This function takes 2 inputs: a non-negative real number x and an initialization value for Newton's method iterations. The nwtsqrt function returns the square root value of x.

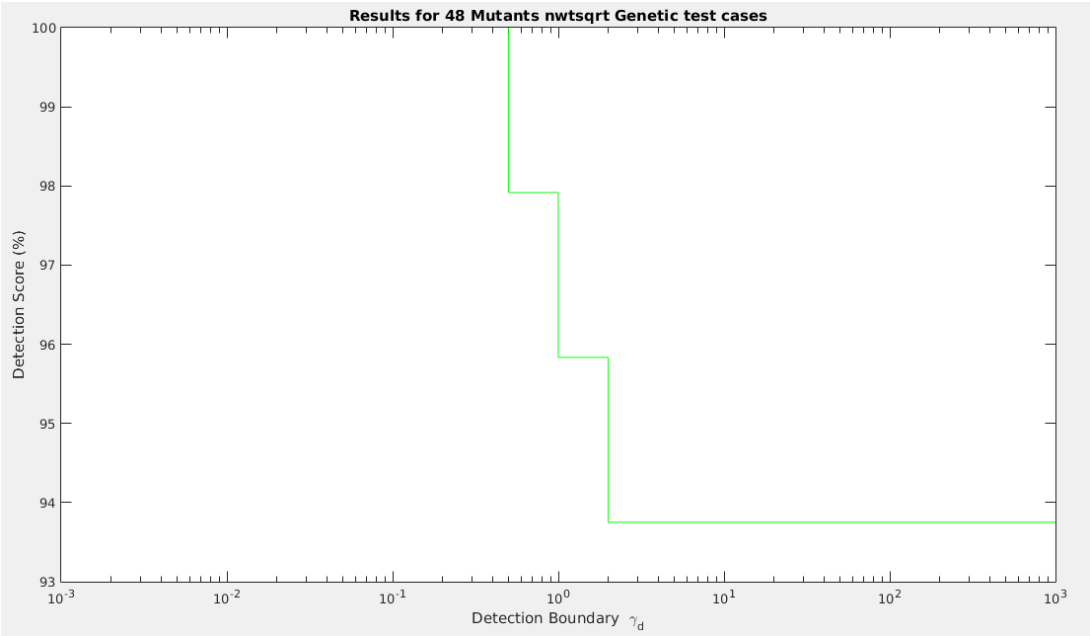
The nwtsqrt detection graph for Hook's test suite can be seen in Graph 13 and the detection obtained from the auto generated test suite can be seen in Graph 14. By carefully comparing the graphs the following observations are made:

1. There is no major difference in the graphs of Tauto and Hook's test suites. The Tauto graph is close to the Tcmb graph. However, the Tauto graph is generated for 48 mutants whereas Hook's graph is generated for 47 mutants.

2. Hook's test suites discovered 47 mutants out of 65. The Tauto test suite detected 48 mutants out of 65. Tauto detected one more mutant than Hook's test suites.



Graph 13: nwtsqrt detection graph generated using Hook's test suite



Graph 14: nwtsqrt detection graph generated using auto generated test suite

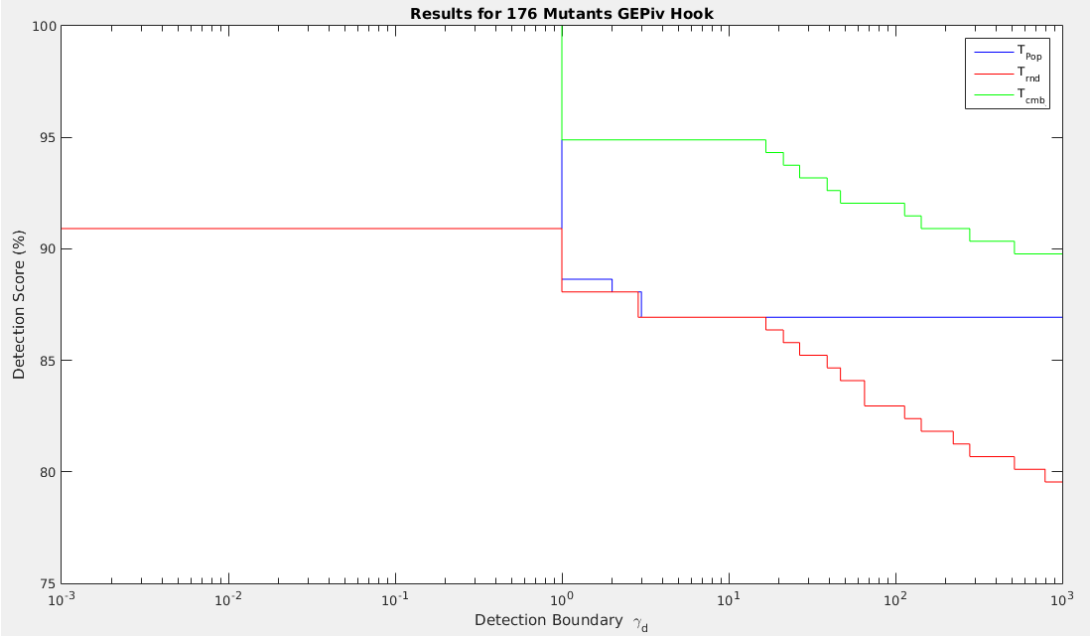
GEPiv

This function solves the system Ax=b, where the first input is A (coefficient matrix), and the second input is b (right-hand side vector). The GEPiv function returns a vector x such that Ax=b is satisfied.

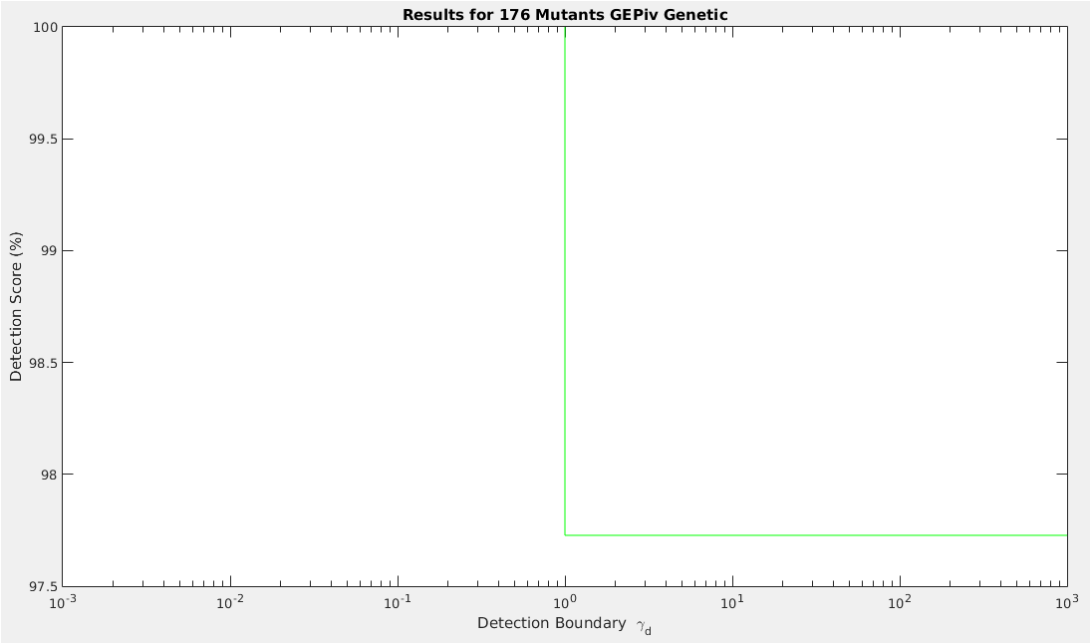
The GEPiv detection graph for Hook's test suite can be seen in Graph 15 and the detection obtained from the auto generated test suite can be seen in Graph 16. By carefully comparing the graphs the following observations are made:

1. The detection score scale for the graphs are different. The detection scale of the Tauto graph is very high when compared to Hook's graph. It can also be clearly observed that mutants got detected at high relative error for the Tauto test suite.

2. There is no difference in the number of mutants detected. Both Hook's test suites and Tauto detected 176 mutants out of 225.



Graph 15: GEPiv detection graph generated using Hook's test suite



Graph 16: GEPiv detection graph generated using auto generated test suite

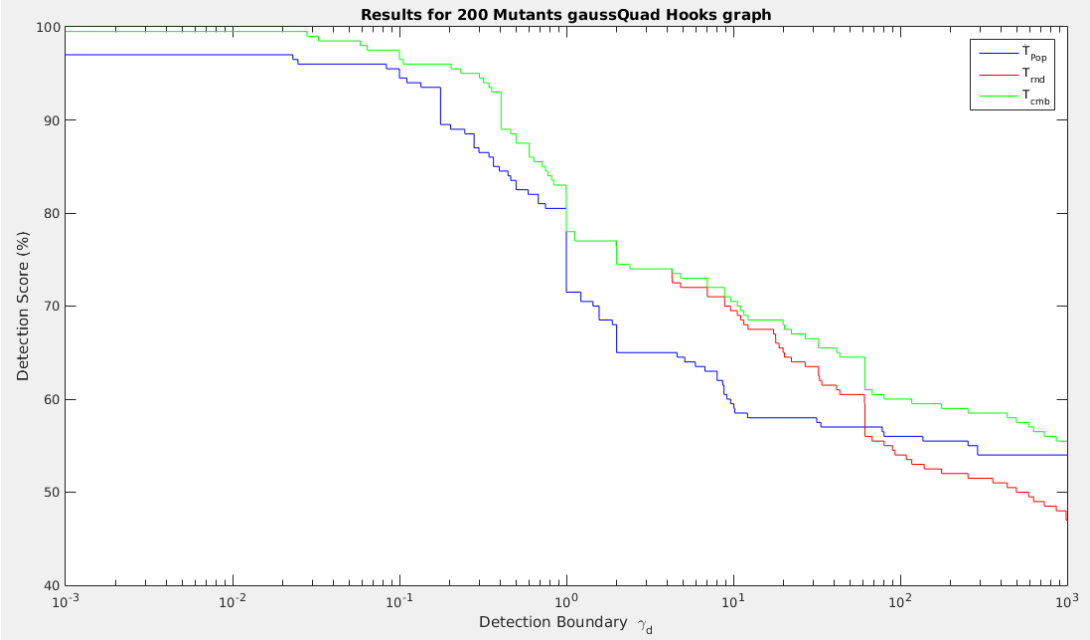
gaussQuad

This function takes 5 inputs: a mathematical function, upper bound of integration, lower bound of integration, the number of panels for integration, and the number of nodes in each panel. Using these inputs, gaussQuad integrates the mathematical function and returns an approximate value.

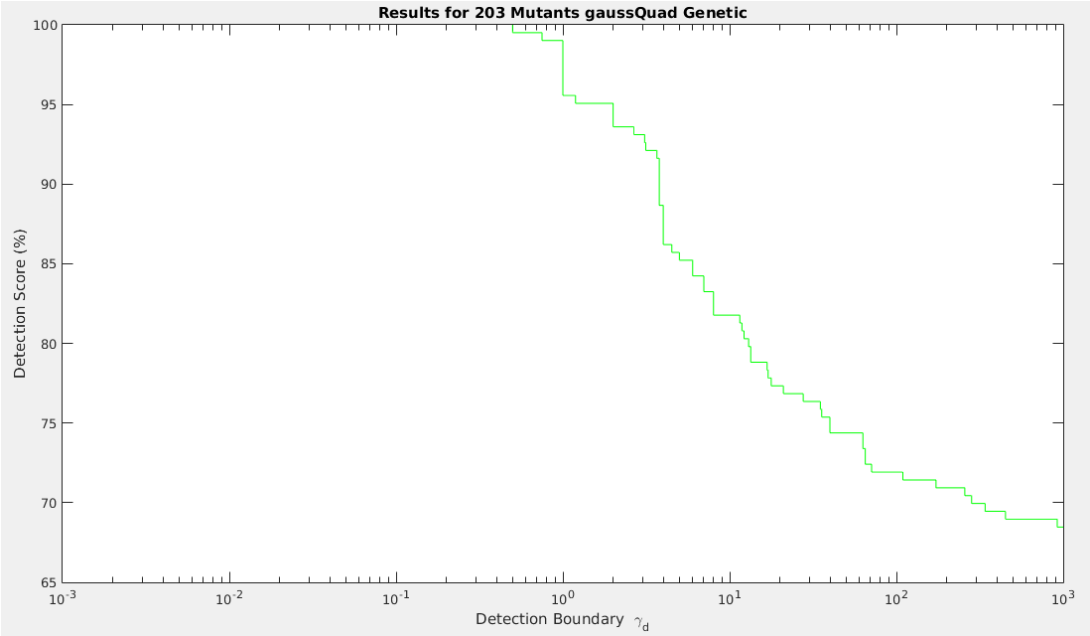
The gaussQuad detection graph for Hook's test suite can be seen in Graph 17 and the detection obtained from the auto generated test suite can be seen in Graph 18. By carefully comparing the graphs the following observations are made:

1. The detection score scale for the graphs are different. The detection scale of the Tauto graph is very high when compared to Hook's graph. It can also be clearly observed that mutants got detected at high relative error. The 100% mutants detection point is also moved to the right in the Tauto graph.

2. Hook's test suites discovered 200 mutants out of 266. The Tauto test suite detected 203 mutants out of 266. 3 extra mutants are detected by Tauto.



Graph 17: gaussQuad detection graph generated using Hook's test suite



Graph 18: gaussQuad detection graph generated using auto generated test suite

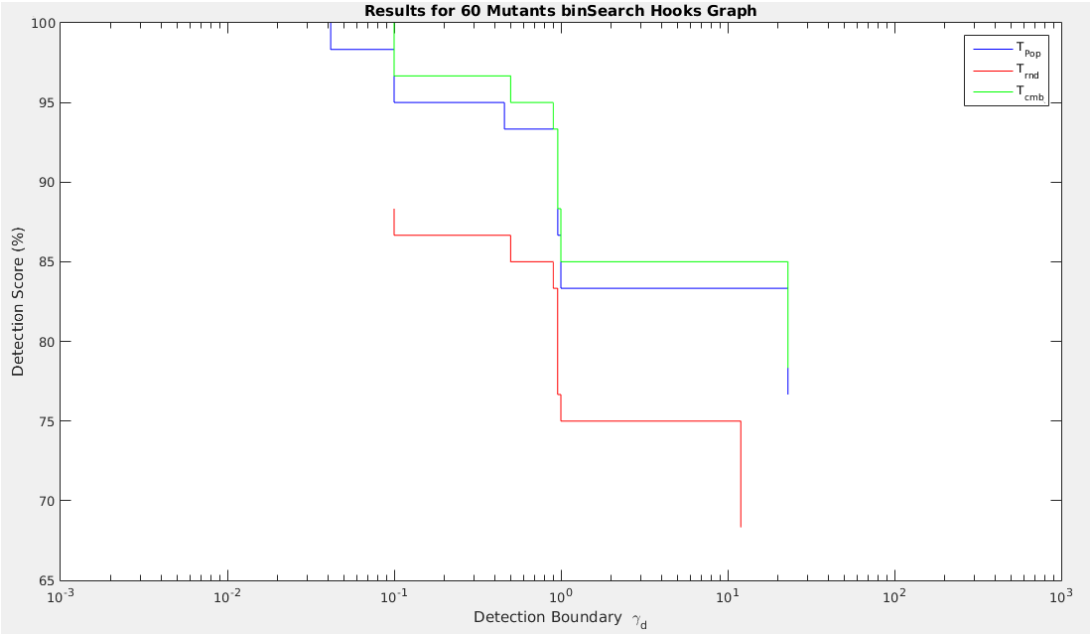
binSearch

This function takes 2 inputs: a vector X with increasing order of real values, and the real value r that is to be located. The binSearch function gives the index value of r in vector X.

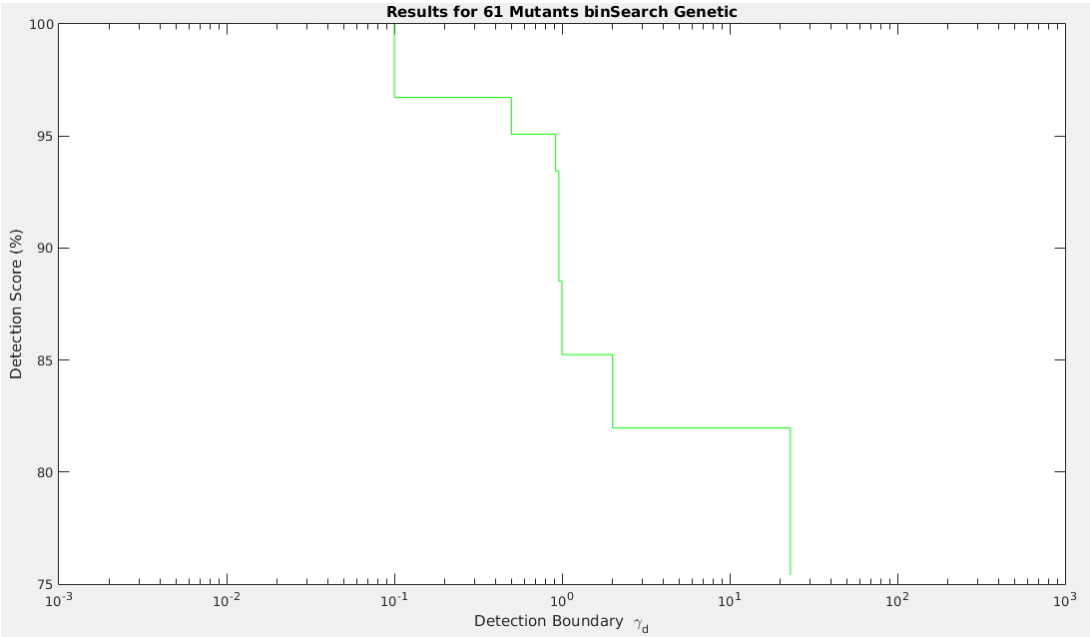
The binSearch detection graph for Hook's test suite can be seen in Graph 19 and the detection obtained from the auto generated test suite can be seen in Graph 20. By carefully comparing the graphs the following observations are made:

1. There is not much difference in the mutants detected between Tauto and Hook's test suites. This might be because of the output that binSearch returns. The binSearch function returns an index value in the range of 0 to 25. High relative errors cannot be obtained, for such values.

2. Hook's test suites discovered 60 mutants out of 82. The Tauto test suite detected 61 mutants out of 82. 1 extra mutant is detected by Tauto.



Graph 19: binSearch detection graph generated using Hook's test suite



Graph 20: binSearch detection graph generated using the auto generated test suite

5.1 Comparison between Hook's test cases and the auto generated test cases

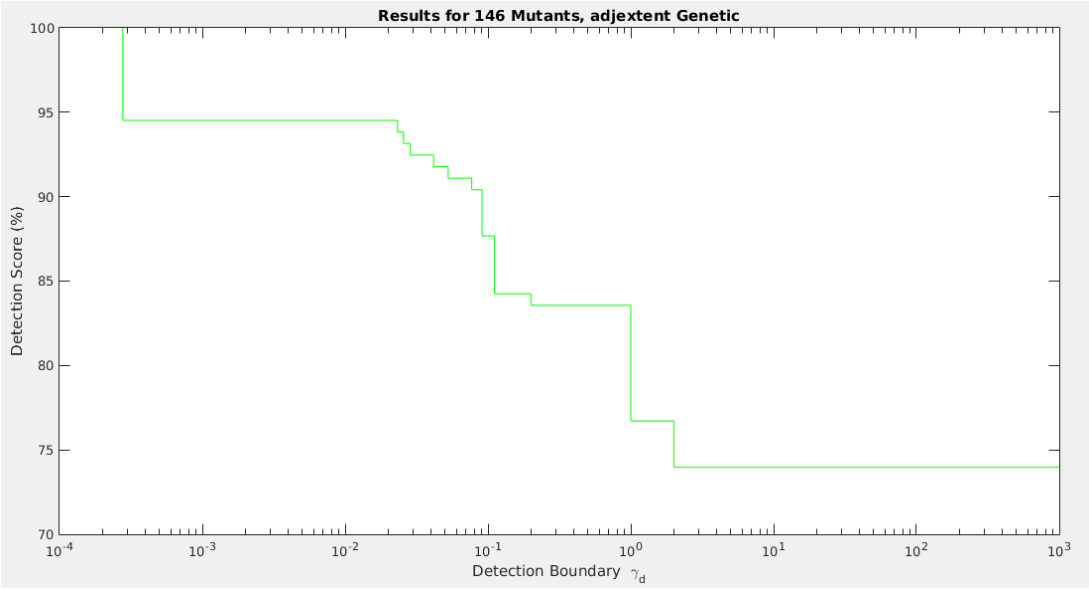
Function Name	Number of mutants	Mutants detected by Hooks test suites	Mutants detected by the auto generated test suites	Difference between mutants detected
sphereFnet	217	197	195	-2
simpson	181	133	136	3
powerit	32	14	20	6
odeRK4	424	362	375	13
nwtsqrt	65	47	48	1
GEPiv	225	176	176	0
gaussQuad	266	200	203	3
binSearch	82	60	61	1
Total	1492	1189	1214	25

Table 5: Comparison between Hook's and Tauto test suites

From Table 5, we can conclude that the auto generated test suites detected more mutants than the manually chosen test cases. Some mutants which are considered equivalent or failed at terminal by Hook's test suites are detected by the auto generated test suite Tauto. Also the auto generated test suites detected the mutants at higher relative error almost for all the functions.

5.2 Generating test cases for a function in LUCY package

To check whether generating test cases works for functions in large scientific projects, I took a function named adjextent from the project "Large scale Urban Consumption of energy (LUCY)". This function takes 5 inputs. They are minimum latitude, maximum latitude, minimum longitude, maximum longitude, and resolution. This function converts latitude and longitude such that it is divisible by input resolution. The following is the detection graph for the function adjextent.



Graph 21: adjextent detection graph generated using auto generated test suite

6 Conclusions and Future Work

6.1 Conclusions

As scientists find it hard to predict a tolerance value for the code under test, we proposed an idea to generate test cases that detect mutants at highest possible relative error. These test cases are generated automatically using a genetic algorithm. This process of generating automatic test cases yielded better results when compared to some manual test cases. Also, the automatic test cases detected more mutants than the manual test cases.

After generating a test suite, scientists need an oracle to perform unit testing on the code. We proposed to create an oracle using independent technologies such as Java and R. There might be some round off and acknowledged errors introduced by built-in MATLAB functions, so to take no chances, the MATLAB function is again implemented in either Java or R, and then unit tested. Once the oracle is implemented, it can also be used while performing regression testing on the code.

6.2 Future work

There are three key areas for future work:

1. There are other optimization algorithms which are more effective than the genetic algorithm. I briefly tried some different optimization algorithms such as global search, multi start, particle swarm, and pattern search. Almost all of these

algorithms gave results better than the genetic algorithm. Because of some customization problems, I had to go with the genetic algorithm.

2. In this research, one test case was generated for each mutant. This results in a large test suite. Research can be done to find a small subset of good test cases from the large test suite.

3. The whole process of mutation testing is computationally very costly, and distributed computing may be applied to make it work faster.

Bibliography

[1] Ma, Yu-Seung, Yong-Rae Kwon, and Jeff Offutt. "Inter-class mutation operators for Java." Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on. IEEE, 2002.

[2] Black, Paul E., Vadim Okun, and Yaacov Yesha. "Mutation operators for specifications." Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on. IEEE, 2000.

[3] Jia, Yue, and Mark Harman. "An analysis and survey of the development of mutation testing." Software Engineering, IEEE Transactions on 37.5 (2011): 649-678.

[4] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In ICSE ’05: Proceedings of the 27th international conference on Software engineering, pages 402–411, New York, NY, USA, 2005. ACM

[5] Y. Jia and M. Harman, “MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language,” in Proceedings of the 3rd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART’08). Windsor, UK: IEEE Computer Society, 29-31 August 2008, pp. 94–98.

[6]A. Simao, J. C. Maldonado, and R. da Silva Bigonha, “A Trans-formational Language for Mutant Description,”Computer Languages,Systems & Structures, vol. 35, no. 3, pp. 322–339, October 2009.

[7] A. J. Offutt, P. Ammann, and L. L. Liu, “Mutation Testing implements Grammar-Based Testing,” in Proceedings of the 2nd Workshop on Mutation Analysis

(MUTATION’06). Raleigh, North Carolina: IEEE

Computer Society, November 2006, p. 12.

[8] Mathur, Aditya P. "Performance, effectiveness, and reliability issues in software testing." Computer Software and Applications Conference, 1991. COMPSAC'91., Proceedings of the Fifteenth Annual International. IEEE, 1991.

[9]Offutt, A. Jefferson, Gregg Rothermel, and Christian Zapf. "An experimental evaluation of selective mutation." Proceedings of the 15th international conference on Software Engineering. IEEE Computer Society Press, 1993.

[10] King, Kim N., and A. Jefferson Offutt. "A fortran language system for mutation-based software testing." Software: Practice and Experience 21.7 (1991): 685-718.

[11] Offutt, A. Jefferson. "Investigations of the software testing coupling effect." ACM Transactions on Software Engineering and Methodology (TOSEM) 1.1 (1992): 5-20.

[12] Madeyski, Lech, et al. "Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation." Software Engineering, IEEE Transactions on 40.1 (2014): 23-42.

[13] Polo, Macario, Mario Piattini, and Ignacio García-Rodríguez. "Decreasing the cost of mutation testing with second-order mutants." Software Testing, Verification and Reliability 19.2 (2009): 111-131.

[14] Jia, Yue, and Mark Harman. "Higher order mutation testing." Information and Software Technology 51.10 (2009): 1379-1393.

[15] Nguyen, Quang Vu, and Lech Madeyski. "Problems of Mutation Testing and Higher Order Mutation Testing." Advanced Computational Methods for Knowledge Engineering. Springer International Publishing, 2014. 157-172.

[16] Hook, Daniel, and Diane Kelly. "Mutation sensitivity testing." Computing in science & engineering 11.6 (2009): 40-47.